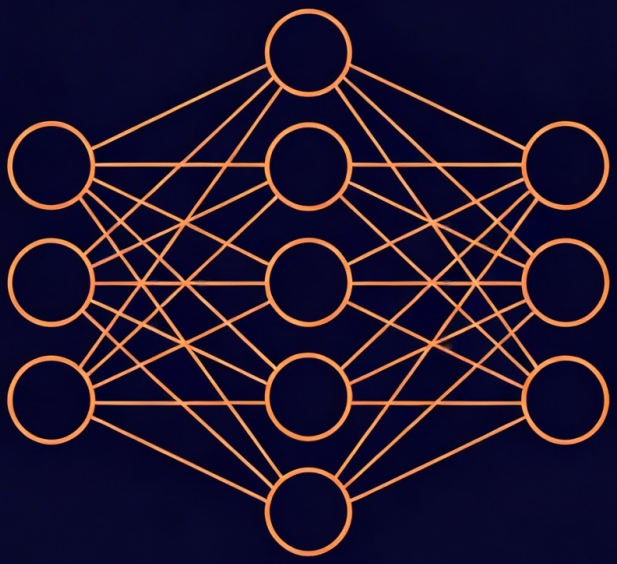
The LLM Field Manual

A Conceptual Guide to How Large Language Models Actually Work



Marcus Hale

The LLM Field Manual

A Conceptual Guide to How Large Language Models Actually Work

by Marcus Hale

CONTENTS

I	What an LLM Is (and What It Isn't)
II	Tokens: The Actual Unit of Computation
III	Embeddings: Turning Words into Vectors
IV	The Transformer: Attention in Plain English
V	Pre-Training: Where the Knowledge Comes From
VI	Instruction Tuning and RLHF
VII	Why Models Hallucinate: The Honest Mechanism
VIII	Sampling: Temperature, Top-p, and What They Actually Do
IX	Context Windows and How They're Really Used
X	Tool Use and Function Calling
XI	Reasoning Models and Chain-of-Thought
XII	Multimodal: Vision, Audio, and the Rest
XIII	Cost, Latency, and Why 'Just Use the Biggest Model' Is Wrong
XIV	Evaluation: How Do You Actually Know a Model Is Better?
XV	The Current Model Landscape in 2026
XVI	Where This Is Going Next (and Where It Isn't)

CHAPTER I

What an LLM Is (and What It Isn't)

Every wrong mental model costs you twice. Once when you build something based on it, and once when the thing breaks and you can't figure out why.

Large language models have generated more wrong mental models per capita than any technology in recent memory. Developers who work with them daily still reach for metaphors that actively mislead: the model looked it up, the model reasoned through it, the model decided to say that. Journalists write about what AI believes. Product managers ask why the model lied. Even researchers occasionally slip into language that implies a kind of inner life the architecture simply doesn't have.

This chapter is about getting the definition right before you get to anything else. Not as a philosophical exercise. As an engineering prerequisite.

The One-Sentence Definition

A large language model is a statistical system trained to predict the next token in a sequence, given all the tokens that came before it.

That's it. Everything else — the fluency, the apparent knowledge, the code generation, the translation, the occasional flashes of what looks like insight — is a consequence of doing that one thing at enormous scale across enormous amounts of text.

The "large" in large language model refers to two things: the number of parameters in the model (often in the tens or hundreds of billions), and the volume of text used to train it. GPT-3, released by OpenAI in 2020, had 175 billion parameters and trained on roughly 300 billion tokens of text. The models that followed trained on more — in some cases trillions of tokens drawn from web crawls, books, code repositories, and scientific papers. Size isn't the whole story, but scale is what made these systems qualitatively different from their predecessors.

The "language model" part is the mathematically precise bit. A language model assigns probabilities to sequences of tokens. Given the sequence "The cat sat on the", it estimates the probability of every possible next token across its vocabulary — which might be fifty thousand entries. The token "mat" gets some probability. So does "floor", "roof", "couch", "idea", and every other token in the vocab, including punctuation and whitespace. Training is the process of adjusting the model's parameters so that its probability estimates get better — meaning, they more closely match what the actual next token was in the training data. The loss function used is cross-entropy, and minimizing it is equivalent to maximizing the likelihood that the model assigns to the real next token.

Do this across enough text, with enough parameters, and the system learns to produce astonishingly coherent, contextually appropriate, and sometimes genuinely useful text. But it is doing exactly one thing at each step: sampling from a probability distribution over its vocabulary. That's the mechanism. Everything we're going to talk about in this book is downstream of that mechanism.

What It Is Not

Getting the positive definition right is half the job. The other half is clearing away the substitutes — the things an LLM is routinely mistaken for, each of which predicts the wrong behavior and leads to the wrong design choices.

Not a Database

The most common misconception, especially among developers who first encounter LLMs through a chatbot interface, is that the model is a kind of database. Ask it the capital of France, it gives you Paris. Ask it who wrote Hamlet, it gives you Shakespeare. It looks like retrieval. It feels like retrieval. But it is not retrieval.

A database stores discrete records at specific addresses. When you query a relational database for a customer record, the system looks up a row at a memory location and returns it. The record is either there or it isn't. The fidelity is binary. Databases do not hallucinate rows.

An LLM stores nothing at a specific address. What training produces is a set of weights — billions of floating-point numbers — that collectively encode statistical regularities from the training corpus. When you ask it a question, it doesn't look up an answer. It generates a plausible continuation of the prompt, token by token, based on patterns in those weights. The answer Paris emerges because "capital of France" co-occurred with "Paris" so many times in training data that the model assigns very high probability to that token in that context.

This distinction matters the moment you push on it. Databases return the same record every time you query the same key. LLMs can return different outputs on identical inputs if sampling is stochastic (more on that in chapter 8). Databases fail cleanly when the record doesn't exist — they return null or throw an error. LLMs fail by generating something that sounds plausible but isn't true, because plausible and true are not the same constraint. A database with a corrupt record is a bug. An LLM that generates a confidently wrong answer is, in a precise sense, working exactly as designed — it found the most statistically plausible continuation of your prompt, and that continuation happened not to correspond to reality.

The database mental model makes you ask the wrong questions. "Why did the model get this fact wrong?" misframes it. The better question is: "What statistical patterns led the model to produce this particular token sequence?" Those are different investigations with different solutions.

Not a Search Engine

A related misconception is that an LLM functions like a search engine with a nicer interface. You type a question, it searches a corpus, it returns the relevant passage. This is particularly tempting because some deployed LLM products do have search functionality attached — but the search component is a separate system, and when it's removed, the model doesn't become a worse search engine. It becomes something with no search capability at all.

Search engines index documents. They maintain inverted indexes that map terms to document locations, apply ranking algorithms like BM25 or learned relevance models to score matches, and return ranked lists of source documents. The source is always there. You can click through. You can see the original context.

An LLM has no index. It has no document store. It cannot tell you which source a piece of information came from, because there is no source — there is only the weight matrix, and the weight matrix has no provenance tracking. The model will often confabulate sources: it will produce a citation that looks real, formatted correctly, with a plausible journal name and year and author list, and the paper will not exist. This isn't deception in any meaningful sense. It's the next-token predictor doing its job — generating a token sequence that looks statistically like a proper citation.

The practical implication is that you cannot use a base LLM as a retrieval system. When people do Retrieval-Augmented Generation (RAG), they are explicitly providing a search layer — usually a vector database that does actual document retrieval — and feeding the retrieved content into the LLM's context. The LLM's role there is synthesis and reformatting, not retrieval. That distinction matters for debugging, for cost estimation, for quality assessment, and for understanding failure modes.

Not a Reasoning Agent

This one is subtler, and more contested, but worth being precise about.

When you ask an LLM a multi-step math problem and it gets the right answer, it looks like it reasoned its way through. When it produces a step-by-step logical argument, it looks like it's applying rules. When it writes working code, it looks like it understood the specification and solved the problem.

Sometimes these appearances hold up. But the mechanism underneath is still next-token prediction. The model produces tokens that look like reasoning — step-by-step structure, logical connectives, intermediate results — because its training data was full of human-generated text that reasoned in exactly those patterns. It has learned that in contexts like math problems, the statistically appropriate output involves numbered steps and specific syntax. That's not the same thing as running a formal reasoning procedure.

The difference becomes visible at the edges. LLMs fail at arithmetic that would take a first-grader a second on paper, because arithmetic isn't pattern-matching over token sequences — it's a procedure. They fail at logical problems where the formally correct answer is statistically rare in the training distribution. They fail at tasks requiring them to maintain a consistent state across many steps without external scaffolding, because there is no persistent state — just the context window, which we'll examine closely in chapter 9.

None of this means LLMs are useless for tasks that require reasoning. It means you need an accurate model of why they succeed and fail. When a model produces a chain of plausible-looking steps and gets the right answer, the fact that it's plausible is doing real work. Plausible token sequences often are correct. But often is not reliably, and the failure modes of a next-token predictor are different enough from the failure modes of a genuine reasoning system that conflating them will mislead your engineering.

The Anthropomorphization Problem

Language evolved to describe minds. We have extremely good vocabulary for mental states — thinking, knowing, understanding, believing, wanting, feeling — and essentially no native vocabulary for "executing billions of floating-point multiplications across a matrix of learned weights." So we reach for the vocabulary we have. The model thinks you asked a different question. The model knows about that event. The model understands the assignment.

These phrasings are not harmless shortcuts. They predict the wrong behavior.

Consider what "knows" implies. If a person knows that the Eiffel Tower is 330 meters tall, you expect that knowledge to be consistent across contexts, accessible on demand, and robustly integrated with other things they know. If you ask them a week later, they'll give you the same answer. If you rephrase the question, they'll still give you the same answer. Knowledge, for humans, has those properties roughly because it's encoded in a way that's relatively stable and context-independent.

LLMs are neither. The same model that correctly answers "How tall is the Eiffel Tower?" may produce a different number if you ask "What's the height of the Eiffel Tower in feet?" and then convert — not because it's inconsistent about a belief, but because the probability distribution over the answer token sequence shifts when the prompt tokens shift. The model has no ground truth to check itself against. There's no stored fact it retrieves; there's a distribution over outputs conditioned on the input.

This is why "the model knows X" is better replaced by "the model reliably outputs X in contexts that resemble the training distribution for X questions." That's uglier. It's also more accurate, and more useful when X starts producing wrong answers.

"Thinks" is even more dangerous as a framing. It implies a process happening internally — deliberation, consideration, evaluation. An LLM doing a single forward pass is not deliberating. It is applying a fixed mathematical function to an input sequence and producing a probability distribution. No search, no deliberation, no revision. The pass completes, a token is sampled, and the next pass begins with that token appended. There's no thinking happening between the tokens.

(Chapter 11 will explain why chain-of-thought prompting partially changes this picture — by generating intermediate tokens that then become input for subsequent tokens, you create something that starts to look more like stepwise reasoning. But that's a specific technique, not the default behavior, and even then the mechanism is still token prediction, not deliberation.)

"Understands" may be the most misleading of the three. Understanding, in the cognitive science sense, involves representing the meaning of something in a way that supports generalization, inference, and transfer. When a person understands a mathematical concept, they can apply it in novel contexts, recognize analogous structures, explain it different ways. The extent to which LLMs do any of this — and it's a serious research question, not a settled one — is an empirical matter to be measured per task, not assumed from fluent output.

Fluent output is not evidence of understanding. A model trained on enough plumbing texts will produce extremely fluent descriptions of pipe joints, but that doesn't mean it knows what a pipe joint feels like to touch, or that it could diagnose a leak in an actual house. The text is downstream of understanding in human-generated training data. In the model, the text is the output. What's underneath it is weights, not comprehension.

A More Useful Vocabulary

If not "thinks," "knows," and "understands" — then what?

The goal isn't to strip you of descriptive language. It's to give you language that makes more accurate predictions. A few replacements that work in practice:

Instead of the model knows X, try the model produces X reliably in this type of context. This surfaces the conditionality. It also makes the relevant test obvious: how reliable is it, across rephrased inputs, across different positions in the prompt, across model versions?

Instead of the model thinks the question means Y, try given this input token sequence, the model's distribution strongly favors outputs consistent with Y. Clunky, yes. But it immediately raises the right follow-up: what about the input sequence is pulling the distribution in that direction? That's an answerable engineering question.

Instead of the model understands the task, try the model generalizes to this type of task from training. This keeps generalization in the picture — which is real and meaningful — while tying it to the training distribution, which is where the limits live.

You don't have to use this vocabulary in every conversation. But you should be able to translate back to it when something breaks.

Why the Scale Matters

There's a version of the next-token predictor framing that, if you're not careful, sounds like it should produce only trivially predictable text — the word that most often follows "The president of the United States" in news articles. That's not what happens at scale, and understanding why is part of getting the definition right.

The reason scale changes things is compression. When you train a model with billions of parameters on trillions of tokens, the model can't simply memorize co-occurrences. The training data is too large and too varied for brute-force memorization to work — the model has to learn abstractions, regularities, and

generalizations in order to predict well across diverse contexts. The pressure of the training objective — predict the next token accurately, across an enormous distribution of texts — forces the model to extract structure from language that generalizes far beyond any specific training example.

This is the insight underlying the empirical success of large language models, and it was not fully anticipated before training runs at scale validated it. Researchers knew that more data and more parameters helped. They didn't know that the resulting models would exhibit something that looked, functionally, like general language capability — the ability to do tasks they were never explicitly trained on, by virtue of the patterns they'd extracted.

Vaswani et al.'s 2017 paper "Attention Is All You Need" introduced the transformer architecture that made training at this scale tractable, by replacing recurrent networks with attention mechanisms that parallelize efficiently on modern hardware. But the transformer is the substrate. The emergent capabilities are a product of the training scale applied to that substrate. What gets encoded in the weights isn't a list of facts — it's a compressed model of the statistical structure of an enormous slice of human-generated text.

That's simultaneously impressive and limited. Impressive because the statistical structure of human-generated text encodes a tremendous amount of useful information — science, history, code, reasoning patterns, social norms, aesthetic conventions. Limited because it's bounded by what's in the training data, conditioned by the representational biases of that data, and subject to the fundamental constraint that statistical plausibility is not a guarantee of truth or reliability.

The Implications for Developers

If you've been building on LLMs for a while, some of this may feel like philosophical ground-clearing without obvious practical stakes. It isn't. The mental model you carry about what an LLM is shapes every design decision downstream.

Take output reliability. If you think of the model as a database, you expect consistent outputs for consistent queries. You'll be surprised and frustrated by variance. If you think of it as a probability distribution sampled under conditions that shift slightly with every input, you'll build in validation layers, expect variance, and design your system to handle it gracefully.

Take hallucination. If you think of the model as a search engine with bad results, you'll think the fix is better retrieval. Sometimes it is — but not always. If you understand that hallucination is the model producing plausible-but-wrong continuations, you'll look at whether the context provides enough grounding, whether the prompt structure is pulling the distribution in a bad direction, and whether the task is one where plausibility and correctness are sufficiently correlated to trust.

Take failure modes on reasoning tasks. If you think the model reasons, you'll assume failures are data gaps — "it just doesn't know this math." If you think the model is doing next-token prediction, you'll recognize that some failures are architectural — the task requires maintaining state or applying procedures that the token-prediction mechanism doesn't support, and no amount of prompting will fix it. The solution is a

different architecture, a tool, or a different task decomposition.

None of this requires you to stop using LLMs. The point isn't that they're less powerful than they appear. On many tasks, they're more powerful than most people expect. The point is that they're powerful in specific ways, limited in specific ways, and understanding which is which is what separates systems that work in production from systems that work in demos.

One Last Distinction

People sometimes hear the "it's just predicting the next token" framing and take it as reductive in a way that dismisses the technology. That's not what's being said.

Predicting the next token well, across the full distribution of human language, is an extraordinarily hard problem. The model that does it well has encoded something genuinely useful about how language works, how concepts relate, how arguments are structured, how code is organized. The mechanism is simple. The result of applying it at scale is not.

The analogy I find useful: natural selection is, mechanistically, just differential reproduction — organisms that leave more offspring have their traits represented in the next generation. That's the whole algorithm. And yet the result of running it for hundreds of millions of years is the nervous system you're using to read this sentence. Mechanism and outcome are not the same thing. Understanding the mechanism does not diminish the outcome. It helps you understand which outcomes are possible and which aren't.

An LLM is a next-token predictor. It is not a mind, a database, a search engine, or a reasoning agent. It is a statistical compression of patterns in human-generated text, able to generate fluent, contextually appropriate continuations of prompts across an enormous range of domains. That's genuinely remarkable. It's also precisely bounded — by its training data, by the context it can see, by the distributional assumptions baked into its weights.

Get the definition right, and the rest of this book will make more sense. Get it wrong, and you'll spend the next hundred pages explaining away surprises that weren't surprising at all — they were just predictions, behaving exactly like predictions.

CHAPTER II

Tokens: The Actual Unit of Computation

Open a browser, paste any paragraph into OpenAI's Tokenizer playground, and the first thing you notice is that the highlights don't line up with words. Some words are a single color block. Others fracture into two or three pieces. A few short function words—"the," "of," "is"—get their own token. But "tokenization" splits into "token" and "ization." "Strawberries" splits into "straw" and "berries." Type something in Arabic or Thai and the color blocks multiply fast, single characters sometimes ballooning into multiple tokens.

That's the actual unit of computation inside every large language model. Not the word. Not the sentence. Not the paragraph. The token.

Everything else in this book flows through that fact. Cost is calculated in tokens. Context windows are measured in tokens. Model speed is measured in tokens per second. The failure modes you'll hit in production—a model losing track of a detail buried deep in a long document, stumbling on a task as trivially easy as counting letters—trace back, more often than you'd expect, to how text was broken apart before the model ever processed it.

What a Token Actually Is

A token is a chunk of text, typically between one character and one word in length. Common English words are usually one token. Long words, rare words, and words in non-English languages tend to fragment into two or more. Numbers, punctuation, and code have their own patterns, which we'll get to.

The vocabulary—the complete set of tokens a model can produce—is finite, fixed at training time, and never changes during inference. GPT-4's tokenizer, called cl100k_base, has a vocabulary of exactly 100,277 tokens. GPT-3 used 50,257. Every piece of text the model reads or writes is encoded as a sequence of integers drawn from that vocabulary. Nothing outside the vocabulary can appear. Everything inside it, combined, can express any text in any language, because the vocabulary always includes individual byte-level fallback tokens that handle characters no BPE merge ever compressed.

The model never works with raw characters or raw words. It works with token IDs—plain integers. The string "the quick brown fox" becomes something like [1820, 4062, 14198, 39935] before the transformer sees it. Those integers get converted to vectors (Chapter 3's territory), and only then does the computation begin. If you want to understand why models behave the way they do, start here: the model's entire world is a sequence of integers, and it produces its responses one integer at a time.

How Byte-Pair Encoding Works

The dominant algorithm for building token vocabularies is Byte-Pair Encoding, almost always abbreviated BPE. It was not invented for language models. Rico Sennrich and colleagues at the University of Edinburgh introduced it for neural machine translation in a 2016 paper, "Neural Machine Translation of Rare Words with Subword Units." The GPT series adopted it, scaled it up, and the field largely followed.

The algorithm starts from the simplest imaginable tokenization: every individual character is its own token. Then it scans the training corpus, finds the most frequently co-occurring pair of adjacent tokens, and merges them into a single new token. It repeats this process—thousands of times—until the vocabulary reaches a predetermined target size.

A small, constructed example makes the procedure concrete. Suppose your entire corpus is:

`` low low low low lower lower newest newest newest widest widest ``

Starting from individual characters, the most frequent adjacent pair is probably "l" and "o," which co-occur six times across "low" and "lower." Merge them into "lo." Now "lo" and "w" are the most frequent adjacent pair in the updated encoding. Merge: "low." Keep going. Eventually "low" becomes one token. "Lower" becomes ["low", "er"]. "Newest" becomes ["new", "est"]. The algorithm has discovered morphological structure—prefixes, suffixes, stems—purely from counting, without ever being told what a suffix is.

Run the same process on a corpus of hundreds of billions of tokens with a vocabulary target of 100,000, and the result is a set of tokens where essentially all common English words are single units, and rare or long words split at their most statistically informative seams. The algorithm isn't clever about meaning. It doesn't know that "un-" is a negation prefix or that "-tion" is a nominalization suffix. It just noticed that those character sequences co-occur in predictable ways across billions of documents, and compressed them accordingly.

BPE isn't the only game. BERT and its descendants use WordPiece, which selects merges by maximizing the likelihood of the training corpus under a language model rather than picking the most frequent pair. Meta's Llama models and Google's T5 use SentencePiece, a library that implements BPE or an alternative called the Unigram Language Model, and which treats whitespace as just another character rather than a word separator—a detail that matters for languages that don't use spaces to separate words. The practical differences between these algorithms show up at the margins, particularly for code and non-English text. But the core logic is the same across all of them: break text into subword units; control vocabulary size; let frequency in data drive the compression.

Walking Through a Real Sentence

Take the sentence: "The transformer architecture tokenizes surprisingly well."

Using cl100k_base, you get roughly:

| Text chunk | Approx. token | |---|---| | "The" | 791 | | " transformer" | 47622 | | " architecture" | 9483 | | " token" | 4037 | | "izes" | 4981 | | " surprisingly" | 29806 | | " well" | 1664 | | "." | 13 |

Eight tokens for a seven-word sentence. Note that space characters are typically absorbed into the token that follows them—" transformer" is one token, not " " plus "transformer." The word "tokenizes" split into "token" and "izes" because "tokenizes" as a complete unit is rare enough that BPE never compressed it fully, while its components appear often enough to be worth their own slots in the vocabulary.

Now try a sentence with lower-frequency vocabulary: "The cryptocurrency exchange liquidated seventeen overleveraged positions."

That same seven-word structure probably runs to 14–16 tokens, because "cryptocurrency," "liquidated," and "overleveraged" all fragment. "Cryptocurrency" in many tokenizers becomes ["crypto", "currency"] or ["crypt", "ocurrency"]—the exact split depends on which BPE merges ran first during vocabulary construction. "Overleveraged" might become ["over", "lever", "aged"] or ["overlever", "aged"] or something else. You need to check. You cannot reliably predict subword splits by intuition.

Non-English text fragments far more aggressively. A sentence of ten everyday Chinese words might require 30–40 tokens, because Chinese characters are written without spaces and were less represented in the English-dominated pretraining corpora of early models. Japanese, Arabic, Korean, Thai, Hindi in Devanagari script—all tokenize substantially less efficiently than English for the same semantic content. The information density per token is lower, which means these languages cost more to process and historically received less effective training signal per character. This is a concrete, structural disadvantage. It's not abstract bias; it's arithmetic.

Code has its own tokenization pathology. Indentation matters in Python, and each space is a character, and groups of spaces may or may not compress depending on context. Long identifiers like getUserAccountBalanceByIdOrThrow fragment into several tokens. Operator-heavy expressions in languages like C++ or JavaScript can be dense with punctuation tokens. A 1,000-line file is not a fixed number of tokens—it depends on the language, the coding style, and the specific identifier vocabulary used.

1,000 Words Is Not 1,000 Tokens

This conflation causes real problems in production: miscalculated budgets, context-window overflows, incorrect rate-limit calculations. Pin down the actual numbers.

For standard English prose—running text with common vocabulary—the standard rule of thumb is: 100 tokens ≈ 75 words, or equivalently, 1 token ≈ 4 characters. OpenAI's own do
So 1,000 words of English prose is roughly 1,333 tokens. A 100,000-word novel sits at approximately 133,000 tokens. A 500-word email is about 667 tokens.

That ratio loosens significantly in other contexts:

Code. Source code is more token-dense than prose. A 1,000-line Python file might run 8,000–15,000 tokens depending on how verbose the identifiers are and how much whitespace the style guide requires. Don't estimate code token counts from line counts.

Non-English languages. For languages that tokenize less efficiently, budget 2–4× more tokens for the same semantic content as English. If your application serves users in multiple languages and your cost model assumes English tokenization rates, you're probably undercounting for non-English speakers.

Structured data formats. JSON is notoriously wasteful. Every quotation mark, every colon, every brace, every space is a token or a fragment of one. {"user_id": 12345, "status": "active"} runs to roughly 16 tokens for information that a sentence like "User 12345 is active" expresses in 7. Minifying JSON before inserting it into a prompt—stripping unnecessary whitespace—is a quick, low-effort optimization that often cuts input token counts by 20–30% for data-heavy workloads.

Numbers and identifiers. Long integers fragment unpredictably. "1234567" might tokenize as ["123", "4567"] or ["12345", "67"] or three pieces. Phone numbers, credit card numbers (never send these, but the point stands), tracking codes, UUIDs—all expand into multiple tokens in ways that don't respect numeric significance.

If you are building anything production-facing—a document summarizer, a code assistant, a customer support pipeline—stop estimating in words. Get the actual token counts. Every major LLM API exposes token counts in response metadata. Open-source libraries like OpenAI's tiktoken and Hugging Face's tokenizers let you count tokens offline, in milliseconds, before making a single API call. There is no reason to guess.

The Three Things Token Count Controls

Cost

API pricing for every major frontier model is stated per token, with input tokens and output tokens billed at different rates. Output tokens cost more—generating a token requires a full forward pass through the network, while input tokens can be processed in a more parallelized batch. As a rough order of magnitude for frontier models in the 2025–2026 period: input runs somewhere in the range of $1–$15 per million tokens, output in the range of $5–$60 per million, depending heavily on the model tier. Smaller or distilled models run one to two orders of magnitude cheaper. Exact prices change with every product announcement—don't anchor to any specific number—but the structure is stable.

The implication is direct. A task that generates 3,000 tokens of output is ten times as expensive as the same task generating 300 tokens, regardless of how intellectually difficult the task is. Prompting strategies that elicit concise, structured responses; routing simple queries to smaller models; setting aggressive max_tokens limits—these aren't premature optimization. They're cost management with a tight feedback loop.

Speed

Models generate output strictly one token at a time, left to right, with no ability to parallelize generation. The time it takes to receive a complete response scales directly with the output token count. A 500-token response takes roughly five times as long to generate as a 100-token response, holding everything else constant.

This is why streaming is nearly universal in production chat applications. Waiting for all 500 tokens to generate before displaying anything leaves the user staring at a blank screen for several seconds. Streaming the tokens to the UI as they arrive lets the user start reading the first sentence while the model is still writing the fifth. Perceived latency drops dramatically even when actual total generation time is identical.

Token generation speed is also the headline spec that matters for latency-sensitive applications. When you see "120 tokens per second" in a benchmark, that's the metric that predicts how long a 600-token response will take: five seconds. Not throughput, not FLOP/s—tokens per second is the practical unit.

Context Windows

A model's context window is the maximum number of tokens it can process in a single forward pass—input prompt plus output combined. Think of it as working memory. Anything outside the window is invisible to the model.

GPT-2 shipped with a 1,024-token context. GPT-3 expanded to 2,048, then 4,096. GPT-4 launched at 8,192 and later moved to 128,000. Claude 3 introduced 200,000-token contexts. By 2026, million-token contexts exist in limited form for specific use cases. The window has grown enormously—but growth in window size does not translate linearly into growth in effective performance, a problem Chapter 9 addresses at length.

What matters here: the context limit is a hard ceiling. Exceed it and the API returns an error, or silently truncates your input in some implementation-defined direction. Either way, your application behaves unexpectedly. Knowing your token counts, before the request is sent, is how you avoid that.

Where Token Boundaries Break Things

The fragmentation isn't just an efficiency question. It creates specific failure modes that look mysterious until you know what's happening.

Letter Counting

"How many r's are in the word strawberry?" This question made the rounds in early GPT-4 testing as a famous example of a simple failure. The answer is three—s-t-r-a-w-b-e-r-r-y. Early GPT-4 would frequently answer two.

The reason is tokenization. "Strawberry" in cl100k_base tokenizes as ["straw", "berry"]. The model has never encountered the individual characters of the word in sequence—they arrived as two opaque subword units. When asked to count the letter r, it's working from statistical patterns about what follows the token "straw" and what follows "berry," not from direct character inspection. Its answer is a guess informed by co-occurrence statistics, not a count.

More recent models do substantially better on this specific example, largely through chain-of-thought training that encourages spelling things out letter by letter before answering (Chapter 11 covers why that helps). But the underlying mechanism hasn't changed. The tokenizer still splits "strawberry" the same way. The

improvement is a learned workaround layered on top of an unchanged representation. You should know the difference.

Hyphenated Words and Structured Strings

"State-of-the-art" tokenizes as something like ["state", "-", "of", "-", "the", "-", "art"]—seven tokens for one compound modifier. That fragmentation affects how the model processes the compound's semantics, but it has a more practical impact on tasks that require verbatim reproduction of structured strings.

Email addresses, URLs, phone numbers, stock ticker symbols, API keys—all fragment into multiple tokens. user.firstname@company.org might become 12 or 14 token fragments. When you ask a model to extract all email addresses from a document and return them unchanged, it's reassembling those addresses from fragments at generation time. It usually works. But "usually" is not "always," and the failure mode is plausible-looking output that is subtly wrong: a hyphen dropped, a dot moved, a subdomain truncated. If you need verbatim extraction of structured strings, validate the output against a regex or schema. Do not assume reconstruction fidelity.

Arithmetic on Long Numbers

The number 847,291 is probably three tokens. The number 5,293,847,102 is more. When you ask a model to add two seven-digit numbers, it is not adding two numbers. It's performing a completion that must reconstruct what the numbers are from their fragmented encodings, track carries across token boundaries that don't align with digit positions, and produce output fragments that reassemble into the correct integer. That's a lot to ask of a pattern-completion system.

Models that use code execution tools to handle arithmetic—writing a quick Python expression and running it rather than computing in-context—solve this cleanly. Chapter 10 covers that architecture. The lesson here is that when you see a model fail at arithmetic on large numbers, the diagnosis isn't "the model doesn't understand addition." It's that you've asked it to do arithmetic in a representation that makes addition unnecessarily hard.

A Subtler Issue: Vocabulary Size Trade-offs

Why not make the vocabulary much larger? If 100,277 tokens compresses well, surely 500,000 tokens would compress better—more words encoded as single units, less fragmentation, shorter sequences.

The trade-off cuts both ways, and it's been studied carefully. A larger vocabulary requires a larger embedding table: one learned vector per vocabulary entry (Chapter 3 explains why). A 500,000-token vocabulary means a much larger matrix to train and store, and—more importantly—means each token appears less frequently in the training corpus. A token that appears only a million times in 500 billion tokens of training data is poorly learned. The model has simply seen less of what it means and how it behaves than a token that appears ten billion times.

Going smaller creates the opposite problem. A 10,000-token vocabulary produces shorter sequences that process faster, but almost everything fragments, the embedding table can't capture enough nuance, and common words get split in ways that lose morphological information the model needs.

The range that's prevailed—30,000 to 100,000 tokens—is an empirically validated middle ground. Common words become single units. Rare words fragment at meaningful seams. Every token in the vocabulary appears often enough in training that the model develops a clear statistical sense of how it behaves. You probably don't need to make decisions about vocabulary size yourself—the models you're using have made them for you. But understanding why the numbers sit where they do will help you reason about which models handle edge cases better and why.

Counting Tokens Before You Need To

The habit worth building is counting tokens proactively, before a request goes out, rather than reactively after something breaks or a bill surprises you.

For OpenAI models, the tiktoken library does this offline in microseconds:

``python import tiktoken enc = tiktoken.getencoding("cl100kbase") tokens = enc.encode("The transformer architecture tokenizes surprisingly well.") print(len(tokens))  # 8 print(tokens)       # [791, 47622, 9483, 4037, 4981, 29806, 1664, 13] ``

For Hugging Face models, every pretrained tokenizer exposes the same information through the same interface:

``python from transformers import AutoTokenizer tokenizer = AutoTokenizer.frompretrained("meta-llama/Meta-Llama-3-8B") ids = tokenizer("The transformer architecture tokenizes surprisingly well.")["inputids"] print(len(ids)) ``

Wire this into your prompt construction pipeline. Count input tokens before sending. Instrument output tokens from the response metadata. Sum them for cost attribution. Set hard max_tokens limits that reflect what your use case actually needs—not the model's maximum, not a round number you picked arbitrarily, but the ceiling above which additional output provides no value for your application.

This discipline turns an opaque API call into a predictable unit of engineering. You'll catch context overflows before they happen in production. You'll spot prompts that are using 4,000 input tokens where 400 would do. You'll have the numbers you need to make real model selection decisions, which is a topic for Chapter 13.

Tokens are not glamorous. They're a preprocessing decision made before the model ever runs, an artifact of data engineering rather than machine learning theory. But they shape everything that comes afterward: what the model sees, what it costs to run, how fast it responds, where it fails, and why.

When a model surprises you—with an unexpected failure, an odd behavior on an edge case, a response that seems to misread the input—ask yourself first: how does this input tokenize? A remarkable number of those surprises dissolve under that question. The remaining ones are what the rest of this book is about.

The next step is understanding what happens once a token has its integer ID. Each of those integers gets mapped to a dense numerical vector before the transformer touches it—a translation step where the geometry starts to encode something that looks, at least loosely, like meaning.

CHAPTER III

Embeddings: Turning Words into Vectors

Every token the model processes is a number. Or, more precisely, a list of numbers. Before the transformer architecture can do anything interesting with the word "bank," it has to turn that word-piece into something a matrix multiplication can operate on. That something is an embedding: a vector of floating-point values, one per dimension, living in a space that might have 768 dimensions, or 1024, or 4096, depending on the model.

This sounds abstract. It isn't.

When a tokenizer splits your input into pieces and assigns each piece an integer ID — token 2891 for "bank," say, or token 4053 for "river" — those integers are just arbitrary labels. There's nothing in the number 2891 that captures what "bank" means, nothing that connects it to "finance" or "loan" or "branch." The embedding layer's job is to translate those arbitrary IDs into vectors that carry actual semantic content.

The translation works through a lookup table. Imagine a massive spreadsheet: one row per token in the vocabulary (GPT-2 had 50,257 rows; modern models often use vocabularies of 100,000 or more tokens), and one column per embedding dimension. Token 2891 maps to row 2891, which is a sequence of — say — 768 numbers. That sequence is the embedding for "bank." At the start of training, every row in this table is filled with random noise. By the end of training, those rows encode something far more organized.

What 768 Dimensions Actually Means

The first instinct is to ask: why so many? A point in three-dimensional space needs three numbers. A point in 768-dimensional space needs 768. Most of us cannot visualize it, and that's fine — no one can. But the math works identically. Distance, direction, and angle are all well-defined concepts in high-dimensional space, just as they are in the three-dimensional world you move through every day.

The reason models use high dimensions is capacity. Each dimension is a potential axis of variation. Three dimensions can express shape, color, and size. Seven hundred sixty-eight dimensions can potentially express syntax, semantics, register, tense, domain, formality, sentiment, and hundreds of relationships we don't have names for — simultaneously. The model doesn't get handed a list of those dimensions and told what each one should mean. It figures out its own organization, emergently, through training.

That last point matters. There is no engineer at Anthropic or OpenAI who wrote a rule saying "dimension 142 encodes formality." The model discovers useful organization on its own.

The Training Signal: Co-occurrence

How does random noise become a meaningful geometric structure? Through next-token prediction.

During pre-training, the model sees a sentence like "She deposited the check at the bank" and has to predict what comes next — or, depending on the training setup, fills in a masked position. The loss function propagates gradients backward through every parameter in the network, including the embedding table. Over hundreds of billions of such predictions, the embedding vectors for tokens that appear in similar contexts get pulled toward each other. Tokens that appear in very different contexts get pushed apart.

This is the same intuition behind earlier embedding methods like word2vec, developed by Mikolov et al. and published in 2013. In that work, researchers trained a shallow neural network — just a few layers, nothing like a transformer — on the simple task of predicting which words appear near a given target word. The result was a set of word vectors with a remarkable property: the geometry of the vector space reflected the geometry of meaning.

The famous example: take the vector for "king," subtract the vector for "man," and add the vector for "woman." You land very close to the vector for "queen."

> *king " man + woman "H queen*

That equation encodes a relationship. The direction in the vector space from "man" to "woman" also exists, in roughly the same orientation, from "king" to "queen." The model never saw a rule that said gender is a meaningful axis of variation. It discovered that direction because words involving men and words involving women co-occur differently in text, consistently, across millions of examples.

This is the mechanism. Not hand-crafted dictionaries. Not curated rules. Pure statistical geometry extracted from co-occurrence.

Directions Are Semantic Features

Push the analogy further, because it's worth sitting with.

Imagine you plot two vectors in two-dimensional space — a simple x/y plane. The vector for "run" sits at (3, 1). The vector for "ran" sits at (3," 1). The vector for "walk" sits at (1, 1). " 1). The pattern is visible: the y-axis seems to encode tense (positive = pres x-axis encodes something about speed or effort.

Real embedding spaces don't look this clean. The dimensions don't align neatly with human concepts, and the clusters don't separate with perfect clarity. But the underlying principle holds: consistent relationships in language — tense, number, formality, domain — tend to be encoded as consistent geometric directions in the vector space.

Research building on Mikolov et al.'s word2vec work identified dozens of these directions in early embedding models. "Paris is to France as Berlin is to Germany" maps to the same relationship vector as "Tokyo is to Japan." Present tense to past tense encodes consistently across thousands of verb pairs. Singular to plural. Positive to negative sentiment. Not designed in. They fell out of training.

For a developer building on top of an LLM, this matters in a practical way. When you provide context that shifts the meaning of an ambiguous word — "river bank" versus "savings bank" — the model isn't doing a dictionary lookup with disambiguation rules. The embedding for "bank" combined with the embeddings of surrounding context tokens creates a composite representation. The attention mechanism (Chapter 4) allows the model to modulate which aspects of "bank"'s embedding are relevant given "river" versus "savings." The geometry of the embedding space is what makes that modulation meaningful rather than arbitrary.

Static vs. Contextual Embeddings

Here's where transformer embeddings diverge importantly from word2vec.

Word2vec gives every token one vector. Always the same vector, regardless of context. "Bank" gets one point in space. Full stop. This is called a static embedding. Fast and useful for many tasks, but fundamentally unable to represent the fact that "bank" means something different in a sentence about rivers than in one about loans.

Transformers produce contextual embeddings. The input embedding for "bank" — the raw lookup from the table — is static. Token 2891 always starts as the same 768-dimensional vector. But by the time that vector has passed through several layers of attention and feed-forward computation, it has been modified by every other token in the sequence. The representation of "bank" in "she crossed to the other bank" is numerically different, in measurable ways, from the representation of "bank" in "she opened an account at the bank." Same token. Same starting vector. Different final representation.

Vaswani et al.'s original transformer paper describes the embedding layer simply: "We use learned embeddings to convert the input tokens and output tokens to vectors of dimension d_model." That sentence understates what happens next, because the embedding layer only begins the process. Attention heads reshape those vectors as context accumulates across layers.

This is what makes transformer embeddings qualitatively more powerful than the word2vec generation. The static lookup table is just the entry point. What the model does with it — layering context through attention — is where disambiguation actually happens.

The Embedding Table Is Learned, Not Designed

There's a misconception that trips up developers new to this field. The embedding table is not pre-loaded with linguistic knowledge from a dictionary, a thesaurus, or any human-curated resource. It starts random. It ends meaningful, because the training process forces it to become meaningful.

Think about what has to be true for the model to minimize prediction loss. If the embedding for "doctor" and the embedding for "physician" are geometrically distant — pointing in completely different directions in the 768-dimensional space — then everything downstream in the network has to work harder to recognize that these two tokens participate in similar sentence patterns. Gradient descent finds it easier to pull "doctor" and

"physician" close together and let the shared neighborhood do the work. The emergent clustering is a byproduct of optimization, not design.

This is why the model can generalize across vocabulary in ways a lookup table could never manage alone. Encounter a rare technical term the model has seen only a handful of times in training, and its embedding will be imprecise — noisy, underfit. But it's probably in a neighborhood of better-represented terms that share similar grammatical and semantic contexts. The model makes reasonable inferences about the rare term by borrowing from its neighbors.

It also explains a failure mode. Words that appear in highly domain-specific contexts, or that carry very different meanings across domains, can have embeddings that serve both contexts poorly. A model trained predominantly on web text might have an embedding for "ablation" that reflects its medical usage more than its machine learning usage — the term "ablation study," ubiquitous in ML papers, might occupy an awkward position in the vector space, somewhat near experimental methods, somewhat near medical procedures, cleanly representative of neither.

Embedding Dimensions and Model Scale

Dimensionality scales with model size. GPT-2 small, at 117 million parameters, uses 768-dimensional embeddings. GPT-2 XL, at 1.5 billion parameters, uses 1,600. GPT-3, at 175 billion parameters, uses 12,288. The trend is consistent across architectures from different organizations.

Why does this matter? Because the embedding dimension — d_model in the transformer literature — is the fundamental bottleneck of the model's representational capacity at each token position. It determines how much information the model can carry per token as the sequence flows through the network. This number appears repeatedly in transformer architecture diagrams because everything flows through it: the width of the residual stream, the dimension of the attention heads, the size of the intermediate feed-forward layers.

When you see a model described as "d_model = 4096," each token position carries a 4,096-dimensional vector through every layer of the network. That's 4,096 floating-point numbers, updated by every attention head and every feed-forward sublayer, representing everything the model has learned about that token's meaning and role in context. At inference time, that's a lot of computation. It's also where the model's representational power lives.

Positional Embeddings: A Separate Problem

There's a wrinkle. Raw token embeddings encode what a token is, but not where it appears in the sequence. Transformers, unlike recurrent networks, process all tokens simultaneously rather than sequentially. Without additional information, the model has no way to distinguish "the dog bit the man" from "the man bit the dog." Identical tokens. Identical base embeddings. Different meaning.

The solution is positional embeddings: additional vectors that encode each token's position in the sequence, added to the token embeddings before the first layer of the transformer. Vaswani et al. (2017) used a fixed, sinusoidal encoding scheme — sine and cosine functions of varying frequencies, one per position. Nearby positions produce similar vectors. The relative displacement between any two positions can be computed via a linear transformation of their encodings. The model learns to notice proximity and distance.

Modern models often use learned positional embeddings instead — the position encoding is itself a trainable lookup table, optimized alongside everything else. Some architectures designed for very long contexts use more sophisticated schemes like Rotary Position Embedding (RoPE), introduced by Su et al. in 2021, which encodes relative rather than absolute position and generalizes better to sequences longer than those seen in training.

The practical upshot: token position matters, and the model knows it — not through rules, but through the same learned-vector machinery as everything else.

What Embeddings Are Not

The geometric intuitions can seduce you into overclaiming. Worth being direct about the limits.

Embedding geometry is statistical, not logical. The vector for "not guilty" is not simply the vector for "guilty" with a negation applied. Negation doesn't have a consistent geometric representation across all contexts in natural language. "I'm not unhappy" and "I'm happy" are semantically close but syntactically different in ways the embedding geometry captures imperfectly. Models struggle with multi-step negation precisely because the co-occurrence statistics of negated phrases don't cleanly mirror the inverse of their positive counterparts.

Embeddings don't store facts. The embedding for "Paris" places it in a neighborhood of other European capital cities, other proper nouns referring to places, other words appearing in sentences about travel, culture, and history. It does not store the latitude of Paris (48.8566° N), the population of Paris (roughly 2.1 million in the city proper as of 2023), or the fact that the Eiffel Tower was built in 1889. That information — to the extent the model can reproduce it — is encoded in the weights of the attention and feed-forward layers, not in the embedding vector itself. The embedding is a starting point for computation, not a database entry.

And embeddings can fail silently. If two concepts that should be semantically distant happen to appear in similar linguistic contexts throughout the training data, their embeddings will be artificially close. Proper nouns that share grammatical patterns — names of countries, companies, diseases — cluster together in ways that occasionally confuse the model. The geometry is learned from text, not from the world.

Embeddings as Infrastructure

Think of embeddings in the context of the full model architecture as the interface layer between the symbolic world of tokens and the continuous world of linear algebra.

The tokenizer produces integer IDs. The integer IDs are arbitrary labels with no mathematical relationship to each other. The embedding table converts those arbitrary labels into points in a continuous, structured space where mathematical operations — addition, subtraction, dot products, matrix multiplications — produce meaningful results.

Everything the transformer does from that point forward operates in that continuous space. The model can compute "how similar is the current token to every other token in the context?" because similarity has a well-defined geometric meaning: the cosine of the angle between two vectors. It can compute "how much should this token attend to that token?" using dot products of query and key vectors derived from those embeddings. None of that machinery works if the starting representations are arbitrary integers.

This is also why fine-tuning works. When you take a pre-trained model and continue training it on a smaller, domain-specific dataset — medical records, legal filings, Python codebases — the embedding table is among the parameters that update. Vectors shift. Medical terms underrepresented in the original training data get pulled into better neighborhoods. Domain-specific jargon starts to cluster appropriately. The geometric structure the base model built becomes the scaffold on which the fine-tuned model hangs its new organization.

What Developers Actually Need to Know

If you're building with LLMs rather than training them, you interact with embeddings in two main ways.

First, through the model's behavior. Every time a model handles synonyms well, or matches the register of your prompt, or groups semantically related concepts in its output — that's embedding geometry doing work. You can exploit this deliberately. Using consistent terminology throughout a long conversation keeps semantically related concepts in tight clusters in the context. Switching terminology mid-prompt creates unnecessary ambiguity. The model doesn't look up your words; it navigates— actually, let's put it plainly: it computes distances. Your vocabulary choices determine what neighborhood you're computing in.

Second, through embedding APIs. Services like OpenAI's text-embedding-3-small and text-embedding-3-large endpoints, updated in early 2024, let you convert text directly into embedding vectors without generating any output. These vectors are useful for semantic search, clustering, and classification. A 1,536-dimensional embedding from text-embedding-3-small represents an entire paragraph as a point in space. Find other paragraphs nearby using cosine similarity. That's the technology underpinning most retrieval-augmented generation systems: store documents as embedding vectors, convert the user's query to an embedding, find the nearest document vectors, retrieve those documents, and inject them into the context window as additional tokens.

The embeddings produced by standalone embedding models are different from the internal token embeddings inside a generative transformer. Related in spirit — both are learned geometric representations of meaning — but distinct in architecture and purpose. The standalone models are often smaller, trained specifically to produce representations that cluster well for retrieval, optimized for cosine similarity as the terminal

operation. The internal token embeddings in a generative model are the entry point to a much larger computation. Not an endpoint. A beginning.

The Bigger Picture

Start with noise. Apply optimization pressure through billions of prediction tasks. End with a geometric structure that encodes syntax, semantics, and dozens of other regularities drawn from the statistical patterns of human language.

That's what an embedding is.

Not magic. Not comprehension. A learned numerical representation, shaped by co-occurrence statistics, that happens to capture enough structure about language to be genuinely useful. The clustering of "doctor" and "physician" in embedding space isn't the model knowing they're synonyms. It's the model having been forced, repeatedly, to predict the same kinds of tokens in the same kinds of contexts when either word appeared — until optimization pressure made them neighbors.

This grounding matters when you encounter the model's limits. If a concept is poorly represented in the training data — rare languages, specialized technical domains, events after the training cutoff — its embedding will be coarser, more approximate. The geometric neighborhood will be less reliable. The model's behavior in those regions of the vocabulary space will be correspondingly less precise.

It also matters when you're evaluating embeddings for a specific application. "How semantically related are these two pieces of text?" is a question about geometry — but the geometry was built on particular training data, with particular vocabulary, at a particular time. An embedding model trained predominantly on English web text will not reliably cluster terms from Swahili legal filings or dense particle physics preprints. The geometry exists. It just wasn't shaped by that domain.

The next chapter moves from individual token vectors to what happens when those vectors interact: the attention mechanism that lets each token's representation be shaped by every other token in the context. That's where the static, lookup-table beginning of an embedding becomes the dynamic, context-sensitive representation that gives transformers their power. But you can't understand attention without understanding what the tokens bring to it.

A point in a learned high-dimensional space. Built from nothing but prediction tasks and optimization. Encoding more structure about human language than anything that came before.

The embedding table is where meaning begins to have a shape.

CHAPTER IV

The Transformer: Attention in Plain English

The paper arrived on arXiv in June 2017. Eight researchers at Google Brain—Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan Gomez, Aukasz K Polosukhin—published "Attention Is All You Need," and within five years it had accumulated more than 100,000 citations. That is not normal for a machine learning paper. What they described wasn't just a better model. It was a new way of thinking about how sequences of symbols relate to each other.

The title is a deliberate provocation. It announces that everything researchers had spent years building—recurrent networks, convolutional layers, gated memory cells—could be discarded. What's left, just attention, is enough.

This chapter explains what that means. No equations. One concrete sentence, used throughout as a running example. By the end, you'll understand not just what a transformer does but why each piece exists.

The Problem Attention Solves

Before you can understand attention, you need to understand what problem it was invented to fix.

Consider this sentence:

> *"The animal didn't cross the street because it was too tired."*

You read that in about one second. You immediately know that it refers to animal, not street. Streets don't get tired. That inference requires you to do something subtle: when you process the word it, you reach back six positions in the sentence and connect it to animal, skipping over street even though street is closer. You weigh those two candidate referents, decide one makes sense and one doesn't, and move on.

Now ask: how would a machine do that?

The naive approach—the one that dominated NLP for the decade before 2017—was a recurrent neural network. An RNN reads a sequence one token at a time, left to right, and carries a hidden state forward like a running summary. By the time it reaches it, the information about animal has been compressed and partially overwritten by everything that came between them. The further back the relevant context is, the harder it is to preserve. This is the famous vanishing gradient problem, and it's why RNNs struggled with long documents.

LSTMs and GRUs (Long Short-Term Memory and Gated Recurrent Units, developed through the 1990s and 2000s) added explicit memory gates to help. They improved things substantially. But they still processed sequences one step at a time, which meant training couldn't be parallelized across a sequence—you had to

compute position 3 before you could compute position 4. On the massive datasets needed for modern language models, that serialization was a bottleneck.

The transformer ditches the sequential processing entirely. Instead, it looks at every token in the sequence simultaneously and lets each token directly query every other token, regardless of distance. Position 1 can attend to position 47 just as easily as position 46. There's no compression artifact from passing through intermediate steps.

That's the core intuition. The mechanism that makes it work is attention.

Attention: The Voting Mechanism

Here's how to think about what attention actually does.

When the transformer processes the word it in our sentence, it doesn't just look at it in isolation. It asks: to understand this token, which other tokens in this sentence are most relevant? It then constructs a weighted combination of information from across the entire sequence, where the weights reflect relevance.

Vaswani et al. formalize this as three components: a query, a set of keys, and a set of values. The terminology is borrowed loosely from database retrieval, though the analogy is imperfect enough that it sometimes creates more confusion than clarity. So instead, think of it this way.

Every token, at every layer of the network, produces three things:

- A question it wants to ask: "What context should I pay attention to?"
- A label it advertises to others: "Here's what information I have available."
- A package of actual information it can contribute.

When it is processed, its question is compared against the labels of every other token—The, animal, didn't, cross, the, street, because, it, was, too, tired—to produce a score for each. Those scores are converted into weights that sum to one: a probability distribution over the whole sequence. The weights then determine how much of each token's information package to include when building it's updated representation.

In our sentence, animal will end up with a high weight. street will get a lower one. The at position one will get very little.

None of this was programmed by hand. The questions, labels, and information packages are learned during training on billions of examples. The network learned—from exposure to text—that pronouns should attend heavily to nouns, that subject-verb pairs attract each other, that modifying phrases stay anchored to what they modify.

This is a critical point. Attention is not a set of hand-crafted grammatical rules. It's a learned weighting function. The "grammar" emerges from the statistics of language.

Query, Key, Value Without the Algebra

Let's trace through what this looks like concretely, still using our sentence.

Every token enters the attention mechanism as an embedding—a vector of numbers (you'll remember this from the previous chapter). Three separate learned matrices are applied to that vector to produce the query vector, the key vector, and the value vector. Think of those matrices as learned projections that pull out different aspects of the token's information for different purposes.

The query for it encodes something like: "I am a pronoun that likely refers to an animate entity nearby."

The key for animal encodes something like: "I am a noun describing an animate entity."

When you compare those two, the match is high. The key for street encodes something like "I am an inanimate location noun," and the match with the pronoun's query is lower.

The comparison is done by measuring how aligned the query vector and key vector are—in practice via a dot product, but you don't need the mechanics. Just know that more aligned means higher score, and the scores across all positions get normalized into weights.

Those weights are then applied to the value vectors. The value for animal contributes heavily to the updated representation of it. The value for street contributes less. The result is a new representation of it that has been enriched by the context of the whole sentence, with animal baked in at high weight.

That's one attention computation. One token, one layer, one look across the whole sequence.

In practice, every token does this simultaneously. While it is attending to animal, cross is attending to animal and street (verbs care about both subject and object), tired is attending back to animal as its semantic subject, and so on. The attention computations across all positions run in parallel. This is why transformers train so much faster than RNNs on modern hardware—GPUs are very good at doing many matrix operations in parallel, and the transformer's structure exposes that parallelism directly.

Multi-Head Attention: Looking From Multiple Angles at Once

Here's something important. A single attention computation produces one perspective on which tokens relate to which. But language is layered. It might need to attend to animal for coreference resolution, but it might simultaneously need to attend to was for syntactic subject-verb agreement, and to tired to understand what predicate it's part of.

These are different relationships requiring different aspects of each token's representation.

Multi-head attention runs several attention mechanisms in parallel—each with its own learned query, key, and value projections—and then combines the results. In the original transformer paper, the base model used 8 heads; larger models use 16, 32, or more. Each head learns to specialize. One head might learn syntactic dependencies; another might learn semantic similarity; another might learn positional proximity. Nobody assigns those roles explicitly. The heads learn what's useful for predicting the next token.

Think of it like convening a panel of specialists rather than consulting one generalist. The coreference specialist votes for animal when processing it. The syntax specialist looks at the verb phrase structure. The local-context specialist looks at adjacent tokens. Their outputs are concatenated and projected back down to the model's working dimension.

This is why transformers can capture multiple simultaneous relationships without tradeoffs. An RNN processing it at one timestep can carry only one compressed view of what came before. A multi-head attention layer can carry eight or sixteen different views at once.

Positional Encoding: Teaching the Model About Order

There's a problem with everything described so far. Attention looks at all tokens simultaneously and computes pairwise relationships. But it doesn't know the order of the tokens.

If you shuffled the words of our sentence to "tired animal because it didn't street the cross the was," the raw attention mechanism—without any additional information—would compute the same attention weights as it would for the original order. The input vectors are the same tokens, just presented in a different sequence, and there's nothing in the dot-product comparison that distinguishes position 1 from position 7.

Order matters enormously in language. "Dog bites man" and "Man bites dog" contain identical tokens with opposite meanings.

The solution in the original paper is positional encoding: a set of signals added to each token's embedding before attention is computed, encoding that token's position in the sequence. In Vaswani et al.'s implementation, these were fixed sinusoidal functions of position—mathematical patterns that give each position a unique fingerprint. Position 1 gets a different signal than position 2, which gets a different signal than position 200.

Adding those signals to the embeddings means the model can differentiate tokens at different positions even when the tokens themselves are identical. The word the in position 2 has a slightly different representation than the in position 6, and the attention mechanism can pick up on that difference.

Subsequent models replaced fixed sinusoidal encodings with learned positional embeddings—instead of a formula, each position slot gets its own vector that the model learns during training. More recent architectures like the one underlying Llama and GPT-4 use a technique called Rotary Position Embedding (RoPE), which encodes position as a rotation applied to the query and key vectors rather than an additive signal on the

embeddings themselves. This turns out to generalize better to longer sequences. But the underlying motivation is the same: give the model the information it needs to know where in the sequence each token lives.

The Feed-Forward Layer: Where the Model Thinks

Attention solves the routing problem. It lets the model collect relevant context from across the sequence. But collecting context isn't the same as doing something with it.

After each attention step, every token's representation passes through a feed-forward sublayer—a small, independent neural network applied identically to every position. It's two linear transformations with a nonlinear activation function between them. In the original paper, the inner dimension of this layer is four times the model's working dimension; if the model works in 512-dimensional space, the feed-forward layer expands to 2048 dimensions before compressing back to 512.

What does this layer do? It applies a stored computation to the representation collected by attention. Think of attention as gathering; the feed-forward layer is processing. Attention says: "Here's the relevant context for this token." The feed-forward layer says: "Now apply knowledge to it."

Research from 2020 and 2021—particularly Geva et al.'s work on "Transformer Feed-Forward Layers Are Key-Value Memories"—suggests that a surprising amount of factual knowledge is stored in the feed-forward weights specifically. When the model knows that Paris is the capital of France, that knowledge lives in the feed-forward layer, not in the attention mechanism. Attention routes relevant context to the token; the feed-forward layer retrieves what the model knows about that context.

This division of labor isn't absolute—modern interpretability research is complicated—but it's a useful heuristic. Attention is the context-gathering mechanism. Feed-forward layers are something closer to the model's memory of learned patterns.

Both components are wrapped with residual connections and layer normalization. A residual connection means the output of attention or feed-forward is added to its input before moving on, rather than replacing it. This gives the signal a direct path through the network that doesn't have to pass through the transformation, which helps gradients flow backward during training and prevents information from being obliterated at any single layer. Layer normalization stabilizes the scale of activations. Together, these techniques make it possible to train very deep networks—networks with dozens of layers—without the training dynamics collapsing.

The Block, and Why You Stack It

One attention sublayer plus one feed-forward sublayer, connected by residuals and normalization, is a transformer block. A transformer model is just this block, repeated.

GPT-2, the 2019 OpenAI model that surprised people with its writing quality, used 48 blocks. GPT-3 used 96. Models like Claude 3 Opus and GPT-4 are not publicly described in detail, but based on the scaling patterns in Anthropic and OpenAI technical reports, they're understood to use hundreds of blocks with model dimensions in the tens of thousands.

Why does stacking help?

Because each layer refines what the previous layer computed. The first layer might resolve basic syntactic relationships—which tokens are syntactically adjacent, which words form noun phrases. The middle layers might resolve semantic relationships—coreference, thematic roles, entailment. The later layers might compute representations that directly reflect what the model needs to predict the next token—meaning, what word comes next in this kind of discourse, at this level of formality, given these entities.

Nobody designed those layers to do those things. This hierarchical processing emerges from training. Interpretability researchers at Anthropic and elsewhere have done exhaustive probing experiments—training small classifiers to predict linguistic properties from activations at different layers—and the pattern is consistent: syntactic structure tends to be resolved in early layers, semantic content in the middle, task-specific prediction information in the final layers.

The stacking is what gives transformers their reach. A single attention layer lets every token attend to every other token, but the relationships it can compute are limited by what can be expressed in one round of query-key-value comparisons. Stack twelve layers, and each layer can attend to representations produced by the previous layer—representations that already encode relationships from the layer before that. By the time you're in layer ten, the representation of it in our sentence might encode not just "this token is a pronoun" but "this token refers to an animate entity that precedes a negated verb and has a property associated with fatigue." Layer one couldn't hold that. Layer ten can.

Long-Range Dependencies: Why This Matters in Practice

The distance problem that sank RNNs is now, in principle, solved. In a transformer, every token can attend to every other token in one step. There is no degradation of signal over distance. A pronoun in position 47 can attend to a noun in position 3 just as easily as to a noun in position 46.

In practice, this doesn't mean the model perfectly resolves all long-range dependencies. Training teaches the model what to attend to based on patterns in data, and if a particular long-range pattern is rare in the training corpus, the model won't learn to handle it reliably. And—as the next chapter discusses—context windows impose hard limits. But within a context window, the architectural barrier of distance has been removed. That's what's new.

This has practical implications you can observe directly. Feed a modern LLM a long legal document and ask it to summarize the key obligations. It can hold the early definitions in context and apply them to clauses that appear thirty pages later, because the attention mechanism can reach back to those definitions at any layer. Feed an older RNN-based model the same document and it loses the thread.

Or consider code. A function name defined at the top of a 500-line file is directly accessible when the model reasons about a call site at line 480. The model doesn't have to hope that the function name survived 480 compression steps. It attends back to it directly.

What the Encoder-Decoder Structure Was For

The original transformer in Vaswani et al. had two major components: an encoder and a decoder. The encoder processed the input sequence—a sentence in French, say—and produced a rich set of contextual representations. The decoder generated the output sequence—the English translation—one token at a time, attending both to previously generated output tokens and to the encoder's representations of the source.

The paper was about machine translation. That task naturally splits into "understand the source" and "generate the target."

Most of the large language models you interact with today, including the GPT series and Claude, don't use this encoder-decoder split. They're decoder-only models: a stack of transformer blocks that take a sequence of tokens and generate the next one, trained purely on next-token prediction over text. The encoder half turned out to be unnecessary for that objective.

Some models—Google's T5 is the canonical example—kept the encoder-decoder structure. BERT and its descendants use encoder-only architectures, optimized for tasks where you want rich contextual representations rather than generation. But for LLMs as most developers now encounter them, "transformer" means decoder-only transformer. Same blocks, simpler structure.

One other feature distinguishes the decoder: causal masking. During training and generation, a decoder is not allowed to attend to future tokens. When processing position 5, the attention weights for positions 6, 7, 8, and beyond are zeroed out—masked. This makes sense: if the model is learning to predict position 6 from positions 1 through 5, it can't be allowed to peek at position 6 when computing position 5's representation. Causal masking enforces the directionality. The model can only attend backward or to itself, never forward.

At inference, this doesn't feel like a constraint, because future tokens don't exist yet—they're what you're generating. But during training, where the full target sequence is available, the mask is essential.

The Transformer as Infrastructure

Here is the thing that most descriptions of the transformer miss: this architecture is not optimized for one task. It's not optimized for translation, summarization, question answering, or code generation. It's a general-purpose mechanism for building contextual representations of sequences.

That generality is why it took over. Before 2017, NLP had specialized architectures for different tasks—convolutional models for classification, recurrent models for generation, memory networks for question answering. Each had its own training tricks and failure modes. Researchers maintained a zoo.

The transformer collapsed the zoo. One architecture, trained at scale on next-token prediction, turns out to acquire competence across essentially all of these tasks. Vaswani et al. couldn't have known this in 2017—they were solving a translation benchmark. But the architecture's properties made it the right substrate for what came next: massive pretraining.

The next chapter describes what happens when you take this architecture and train it on half a trillion tokens. The transformer provides the machinery. Pre-training provides what, loosely, gets called knowledge.

What You've Just Understood

Step back and take inventory.

You now know that a transformer processes all tokens in a sequence simultaneously rather than one by one. You know that attention is a learned mechanism for computing weighted combinations of information across positions, letting each token decide which other tokens are most relevant to its own representation. You know that multi-head attention runs several of these mechanisms in parallel, each learning to specialize in different kinds of relationships. You know that positional encoding injects sequence-order information so the model knows where each token sits. You know that feed-forward sublayers store and apply learned patterns to the context gathered by attention. And you know that stacking many such blocks lets the model build increasingly abstract, layered representations—so that a pronoun in the tenth layer doesn't just know it's a pronoun; it knows which entity it refers to, what that entity is doing, and what kind of thing might plausibly come next.

None of this requires the model to "understand" language in any cognitive sense. It requires only that the learned weights—trained on billions of examples to minimize prediction error—encode useful patterns. The model doesn't know that animals get tired and streets don't. It knows that in its training data, patterns like animal ... it was too [adjective] were consistently resolved with the pronoun pointing to the animate entity. That's not understanding. It's an extremely powerful statistical structure.

Keep that distinction close. It explains nearly everything that follows.

CHAPTER V

Pre-Training: Where the Knowledge Comes From

Before anyone talks about what a model "knows," you need to understand where that knowledge came from. And the honest answer is both simpler and stranger than most people expect.

Pre-training is the first and most expensive phase of building a large language model. It is the phase that produces a base model — a raw artifact that can predict text but hasn't yet been shaped into an assistant that answers questions helpfully, refuses harmful requests, or follows instructions. Everything that happens after pre-training — instruction tuning, RLHF, fine-tuning — is sculpting a thing that pre-training already built. To understand any of the later chapters, you need a clear picture of what that thing actually is and how it got made.

One Objective, Stated Plainly

The training objective is simple enough to fit in a sentence: given a sequence of tokens, predict the next one.

That's it. That is the entire learning signal for a base model. The model sees some tokens, produces a probability distribution over every possible next token, and gets scored on how much probability mass it assigned to the token that actually came next. The scoring function is cross-entropy loss. When the model assigns high probability to the right answer, the loss is low. When it spreads probability mass across wrong tokens, the loss is high. Training is the process of adjusting the model's parameters — billions of numbers — to make that loss smaller, iteratively, across hundreds of billions of examples.

The optimizer — typically Adam or a variant of it — computes gradients by backpropagating the loss signal through every layer of the network, and nudges each parameter in the direction that would have reduced the loss on that batch. Then the next batch arrives. And the next. For weeks, sometimes months, on clusters of thousands of GPUs running in parallel.

Nothing about this objective mentions facts. Nothing mentions reasoning. There is no explicit reward for being correct about the world, no penalty for confabulation, no signal that says "Paris is the capital of France and you should encode that." What the objective does reward is accurately modeling which tokens tend to follow which other tokens — across every genre, register, language, and domain in the training corpus. The "knowledge" a base model ends up with is a byproduct of that optimization, not its explicit target.

Keep that in mind. It matters enormously for everything else.

What Goes Into the Corpus

If next-token prediction is the engine, the training corpus is the fuel. And the scale of it is worth sitting with for a moment.

GPT-3, published by OpenAI in 2020, was trained on roughly 300 billion tokens. That sounds like a lot until you hear that Meta's Llama 3, released in 2024, was trained on 15 trillion tokens for its larger variants. Fifteen trillion. A typical English word converts to somewhere between 1.2 and 1.5 tokens using a Byte-Pair Encoding vocabulary, so 15 trillion tokens is roughly 10 to 12 trillion words — an amount of text no human being, or collection of human beings, could read in any meaningful sense of the word.

The data comes from several overlapping sources. Common Crawl is the dominant one for most large models: a nonprofit organization that has been crawling the web continuously since 2008 and publishing raw HTML snapshots. As of 2024, Common Crawl's archive exceeds 250 petabytes. Model builders take a filtered, deduplicated slice of it — still measured in trillions of tokens. Then they layer in curated datasets: books (both digitized public-domain libraries and licensed text), academic papers from sources like arXiv, code from GitHub repositories, and Wikipedia in multiple languages.

The filtering matters. Raw Common Crawl includes spam, SEO garbage, duplicated boilerplate, and content of every description. Most training pipelines apply heuristic filters (remove pages shorter than some word count, remove pages that are mostly non-alphabetic characters) and classifier-based filters (a model trained to distinguish high-quality text from low-quality text — an amusing recursive dependency). Then deduplication: the same paragraph appearing on 10,000 different websites should train on once, not 10,000 times, because repeated identical text biases the model to reproduce it verbatim rather than learn from it.

Even after filtering and deduplication, the corpus is deliberately overweighted toward certain domains. Code gets a disproportionate share of most modern training runs relative to its prevalence on the internet, because coding ability appears to transfer beneficially to structured reasoning tasks. High-quality curated text — books, peer-reviewed papers, carefully written Wikipedia articles — gets upsampled relative to raw web text. These weighting decisions are engineering choices made by the people building the model, not properties of the data itself, and different labs make different calls. OpenAI, Anthropic, Google, and Meta have each published varying amounts of detail about their data composition decisions; none has published a full accounting.

One implication worth flagging: whatever gaps, biases, or errors exist in the training corpus, the model absorbs them. A corpus that skews toward English, toward text produced by educated writers, toward content from wealthy countries, and toward information that existed before a certain cutoff date will produce a model that reflects all of those skews. This is not a fixable bug in the training algorithm. It is a direct consequence of training on human-produced text.

The Mechanics at Scale

Here's where the word "scale" stops being a vague compliment and starts being a precise technical problem.

Training a large model requires holding the model's parameters in GPU memory across many machines simultaneously, passing batches of training examples through the forward pass, computing the loss, running backpropagation to get gradients, and synchronizing those gradients across all the machines before updating the parameters. Each of those steps is non-trivial at scale.

GPT-3's 175 billion parameters, stored in 16-bit floating point, require about 350 gigabytes just to store the weights — far more than fits on any single GPU. Training uses model parallelism (splitting the model across multiple GPUs), tensor parallelism (splitting individual layer computations across devices), and pipeline parallelism (assigning different layers to different stages of a pipeline). Getting all of this to run efficiently, without one slow GPU stalling all the others, is a significant systems engineering challenge. OpenAI's 2020 GPT-3 paper noted that the training run itself required on the order of 3.14×10^{23} floating point operations. That number is not very human-legible, so think about it this way: a high-end research GPU doing about 300 teraFLOPS would need roughly 33,000 GPU-days to match that compute budget. Actual training happens on clusters of thousands of such GPUs running in parallel for weeks.

Anthropic, in its public research and model cards, has been less specific about raw compute numbers but has consistently referenced scaling in terms consistent with the broader literature. The training infrastructure for Claude 3 and subsequent models is built on large-scale TPU and GPU clusters across cloud providers.

There is a known failure mode in large training runs: loss spikes. During training, the loss curve — which you're hoping to see steadily decrease — occasionally jumps sharply upward and then (usually) recovers. The causes are not fully understood, but common candidates include particular batches of data with unusual token distributions, numerical instability in certain layer operations, and interactions between the learning rate schedule and the gradient statistics. Teams running large training runs monitor these events closely and in some cases roll back to an earlier checkpoint. It is one reason why training a frontier model is not just expensive but genuinely risky: a large spike late in training can destroy weeks of compute investment.

Scaling Laws: How Much Is Enough?

In 2020, researchers at OpenAI published a paper commonly called the Kaplan scaling laws paper. The central finding was that LLM performance — measured by validation loss on held-out text — follows smooth power-law relationships with three quantities: the number of model parameters, the size of the training dataset, and the total compute budget. Double the compute, and you get a predictable improvement in loss. The relationship held across many orders of magnitude, which was surprising and practically very useful: it meant you could run small experiments and extrapolate with reasonable confidence to larger runs.

But the Kaplan results implied something specific about how to allocate a fixed compute budget. Their recommendation was to scale parameters faster than data — given a fixed budget, make the model bigger rather than training it on more tokens.

This advice turned out to be wrong, or at least incomplete.

In 2022, researchers at DeepMind published what became known as the Chinchilla paper — formally, "Training Compute-Optimal Large Language Models" (Hoffmann et al., 2022). They ran a careful set of experiments varying both model size and training token count across a fixed compute budget, and found that the Kaplan scaling law had systematically underestimated the value of training data relative to model size. Their conclusion: for a compute-optimal run, you should scale parameters and tokens roughly equally. If you have enough compute to train a 10-billion-parameter model for 200 billion tokens, you'd do better training a smaller model for more tokens — or the same size model for longer — than the Kaplan recipe would suggest.

Their demonstration model, Chinchilla, was 70 billion parameters trained on 1.4 trillion tokens. It outperformed Gopher, DeepMind's own earlier model at 280 billion parameters trained on 300 billion tokens, across most benchmarks. Smaller model, more data, better results.

The practical implication spread quickly through the field. Meta's Llama series leaned into it: Llama 2's 7 billion parameter model was trained on 2 trillion tokens, intentionally overtrained relative to the compute-optimal point because smaller models that have seen more data are cheaper to run at inference time. You pay the compute cost of training once; you pay the inference cost every time a user sends a request. A smaller model that matches a larger one's quality is strictly better for a production system.

There is still active debate about where the scaling laws break down, whether different architectures shift the curves, and whether data quality changes the slopes. This is not a solved question. But Chinchilla's central correction — that pre-training data quantity is roughly as important as model size — is now the consensus starting point.

What "Knowledge" Actually Means Here

Here is the reframing that most explanations skip, and it's the one that will save you from a category error that haunts almost every LLM product decision.

When someone says a model "knows" that the Earth orbits the Sun, they are using a metaphor. The model does not have a record in a database that says {"fact": "Earth orbits Sun", "confidence": 1.0}. There is no such structure inside the model. What there is: billions of parameters whose values were shaped, through the training process, by the fact that tokens like "Earth," "orbits," "Sun," "heliocentric," "planet," "solar system," and hundreds of related terms co-occur in the training data in patterns that are consistent with that fact being true.

If you prompt the model with "The Earth orbits the —" it will assign high probability to "Sun" not because it looked that fact up, but because in every physics textbook, every astronomy article, every Wikipedia page, every science homework answer in its training data, that token sequence resolved to "Sun." The model has learned to continue text the way text like that gets continued.

This is a genuinely different kind of "knowing," and it has specific consequences.

First: it explains breadth. A model trained on 15 trillion tokens of diverse text develops internal representations that capture regularities across an enormous range of domains — chemistry, cooking, legal language, JavaScript syntax, Homeric Greek, plumbing. Not because anyone explicitly taught those domains, but because tokens from all of those domains were in the corpus, and predicting them required the model to develop structured internal representations of their logic and vocabulary. The breadth of a frontier model's apparent "knowledge" is not magical. It is the natural result of training on enough diverse text to require learning the underlying regularities.

Second: it explains failure modes. The model has no privileged access to ground truth. It cannot verify that what it generates is accurate. It will assign high probability to the most plausible continuation of the prompt given what's in its context and what it learned during training — and "plausible" and "true" are not the same thing. A confident-sounding falsehood that would fit naturally in a Wikipedia article will score well by the model's internal measure of what should come next. There is nothing in the training objective that punishes this.

Third: it explains why knowledge cutoffs are hard. The model doesn't have an index of facts with dates attached. It has statistical patterns learned from text that was produced before a certain date. When you ask it about something that happened after training ended, it doesn't "know it doesn't know" in any reliable sense — it just generates the most plausible tokens for that context, which might be accurate (if similar patterns existed before the cutoff), confidently wrong (if the model extrapolates from stale information), or an admission of uncertainty (if enough examples of "I don't have information about recent events" appeared in instruction-tuning data, which is a different phase).

The Corpus as Mirror

One implication of this framing that doesn't get enough attention: the model is not a neutral reasoner who happens to have read a lot. It is a distillation of the text it trained on, including all the views, framings, and factual errors in that text.

If Wikipedia has a systematic bias in which scientific topics get detailed coverage versus which get stubs, the model absorbs that disparity. If certain political framings dominate English-language journalism, the model has seen those framings far more often than alternatives. If a particular technical misconception was repeated widely enough in Stack Overflow answers and blog posts, the model may reproduce it with apparent confidence.

This is not a bug in the training algorithm that a better optimizer could fix. It's a feature of learning from human-produced text. The text humans produce reflects the knowledge, biases, errors, and rhetorical habits of the humans who produced it. A model trained to predict that text learns to predict those patterns.

It also means that the model's behavior on specific questions can be quite sensitive to how common certain answers were in the training corpus. A well-known anecdotal observation: models trained before extensive alignment work often reflected whatever the majority view in their training data was, including majority

views on contested questions where the "majority" was determined by which forums and websites got crawled. This is one motivation for the post-training alignment work covered in the next chapter.

What the Model Has and Hasn't Seen

The training corpus is finite and historical. The model was trained on text that existed up to some date, and nothing after that. This creates a hard information boundary that gets called the knowledge cutoff.

Cutoff dates are usually published by the model providers: GPT-4 had a cutoff of April 2023 at release; various Claude models from Anthropic have published cutoffs in their model cards. But the cutoff is messier than it sounds for two reasons.

One: not all text before the cutoff is equally represented. Text from six months before the cutoff has had less time to propagate across the web than text from three years before. An event that happened in February 2023 might appear in far fewer training documents than an event from February 2020, simply because fewer articles, analyses, and references had been written and crawled by the time the corpus snapshot was taken. The model's "knowledge" of recent-to-cutoff events is often sparser and less reliable than its knowledge of older events.

Two: the model doesn't know its own cutoff date with precision. It was not trained with an explicit "you know everything up to date X and nothing after" signal. Its uncertainty about recent events bleeds out gradually rather than stopping cleanly at a line.

Both of these points matter if you're building a system that relies on the model's factual recall for anything time-sensitive. The model can tell you its approximate cutoff date if you ask, but that number should be treated as a rough guide, not a guarantee.

The Base Model Is Not a Product

One thing that confuses people who first encounter base models: they are not chatbots. They are not assistants. They do not have a helpful disposition. They will complete text in whatever direction the text's statistical properties suggest.

If you prompt a raw base model with "How do I make a bomb?" it does not refuse. It continues the text in the most statistically natural way, which might be a chemistry explanation, might be a joke setup, might be a recipe for a gag, might be a gun store query. It depends on how sequences starting with those tokens typically resolved in the training data. There is no value judgment in the forward pass. The training objective never asked for one.

This is not a failure of the base model. It is what a base model is. The safety behaviors, the refusals, the helpful tone, the appropriate disclaimers — all of that comes from post-training, which is an entirely separate process. The base model's job was to learn to predict text. It did that. Now it's a raw material.

This distinction matters because it clarifies what post-training can and cannot accomplish. Instruction tuning and RLHF can steer a capable base model toward useful behavior. They cannot install capabilities that the base model doesn't have. If the base model's training data didn't include enough examples of a certain kind of reasoning, no amount of instruction tuning will reliably produce that reasoning. Post-training works with what pre-training built.

How Long Training Actually Takes

Here's a number that helps calibrate scale: GPT-4's training run, while not fully detailed publicly, almost certainly took months of continuous GPU compute. Llama 3's 405-billion-parameter model required what Meta described as significant compute resources across their GPU infrastructure. These are not experiments you run overnight and inspect in the morning.

The practical consequence for the field is that only a small number of organizations have the capital to train frontier-scale models from scratch. A rough back-of-envelope: training a 70-billion-parameter model on 2 trillion tokens at current cloud GPU pricing runs into the tens of millions of dollars. Training a frontier model at GPT-4 or Claude 3 Opus scale almost certainly runs into the hundreds of millions when you account for failed runs, data infrastructure, and engineering time. This is why most companies building LLM-powered products are working with pre-trained models via APIs or downloaded open-weight checkpoints — not training their own.

That concentration of training capability is itself a fact about the current landscape worth keeping in mind. The base models available to most developers were trained by a handful of organizations with access to resources most companies don't have. Whatever was in those organizations' training corpora, whatever filtering and weighting decisions their data teams made, is embedded in the models everyone else is building on.

What You Should Take From This

The mental model to hold is this: pre-training is an enormous statistical compression operation. You take hundreds of billions of tokens of text — everything from academic papers to Reddit threads to Python code to nineteenth-century novels — and you compress the regularities of that text into the weights of a neural network. The resulting model is extraordinarily good at predicting what text should come next across an enormous range of contexts.

It is not a database. Information is not stored at a retrievable address. It is not a search engine. It cannot look up facts. It is not a reasoner in any formal sense. It has absorbed statistical patterns that allow it to behave, in many situations, as if it knows things and can reason — and often that behavior is genuinely useful. But the mechanism underneath is pattern completion over learned statistics, not fact retrieval or logical inference.

That distinction is not academic. It determines why the model can translate between languages it was never explicitly trained to translate between. It determines why it can write code in a programming language

released a year before its training cutoff. It determines why it hallucinates with complete confidence. It determines which tasks it will generalize to and which tasks will break it.

Everything else in this book is downstream of that. The errors, the successes, the surprising capabilities, the frustrating failures — they all trace back here, to a training objective that asked for one thing and got something much stranger and more interesting in return.

CHAPTER VI

Instruction Tuning and RLHF

Take a base model fresh off pre-training. Feed it the prompt "Explain photosynthesis to a ten-year-old." What do you get?

Not an explanation. You get something like: "Explain photosynthesis to a ten-year-old. Explain mitosis to a twelve-year-old. Explain plate tectonics to a—"

The model continues the pattern because that's all it knows how to do. It has seen thousands of worksheet headers, textbook question lists, and study guide templates in its training data. A prompt that looks like a list item gets completed like a list item. There's no understanding of the intent behind the words. The model is doing exactly what it was trained to do—predict the next most likely token—and a list of questions is a perfectly plausible continuation of a list of questions.

This is the gap. Pre-training produces a remarkably powerful statistical machine with a genuinely broad representation of human language and knowledge. It does not produce a useful assistant. Bridging that gap is the job of post-training, and it happens in two stages: supervised fine-tuning and reinforcement learning from human feedback.

Understanding both stages precisely—what they optimize, what they change, and what they leave untouched—is probably the most practically useful thing you can know about how modern LLMs work. Because almost every behavioral property of a deployed model that matters to you as a developer originates here, not in the architecture.

Stage One: Supervised Fine-Tuning

The first stage is the simpler of the two, and it's worth understanding clearly before you get to the more elaborate machinery.

Supervised fine-tuning, almost always abbreviated SFT, works like this: you take the pre-trained base model and continue training it on a new, curated dataset. This dataset looks different from the pre-training corpus. Instead of raw internet text, it contains prompt-response pairs—an instruction on one side, a high-quality answer on the other.

"Summarize this article in three bullet points." !’ [three excellent bullet poin

"Write a Python function that reverses a linked list." !’ [a correct, well-comm

"I'm feeling really overwhelmed at work. Any advice?" !’ [a warm, practical r
catastrophize]

The model trains on these examples with the same next-token prediction objective it used in pre-training. The difference is what it's predicting: now it's learning to complete responses given instructions, rather than completing arbitrary internet documents. You're not retraining the model. You're nudging its distribution toward a specific behavioral regime.

How many pairs do you need? Fewer than you might think. OpenAI's InstructGPT paper, published in 2022 by Ouyang et al., used roughly 13,000 prompt-response pairs for their SFT stage. That's not a large number by ML standards—it's the kind of dataset a team can assemble with a few dozen contractors working for a few months. The pre-training has already compressed so much linguistic and conceptual structure into the model's weights that SFT only needs to redirect rather than rebuild.

The contractors who write those responses matter enormously. They're not random crowdworkers. OpenAI's InstructGPT paper describes selecting contractors who scored highly on a test evaluating their ability to identify sensitive content and write careful, nuanced answers. Anthropic has discussed similar labeler selection criteria in their public documentation. The quality distribution of the SFT dataset essentially defines a ceiling for what the resulting model can aspire to—you cannot fine-tune your way to better-than-human responses using only human-written demonstrations.

After SFT, the model behaves significantly more like an assistant. Ask it to explain photosynthesis to a ten-year-old and it will try. It'll follow formatting instructions, adopt a requested persona, produce structured output. It has learned the shape of the interaction.

But SFT alone has a problem. The model has learned to imitate the demonstrations it was trained on. When a prompt resembles something in the training set, it does well. When it doesn't—edge cases, unexpected phrasings, novel tasks—behavior degrades. Worse, imitation carries no inherent quality signal. The model doesn't know which responses are better, only which responses were included. It can't generalize the underlying preference. That's what the second stage is for.

Stage Two: Building a Reward Model

Before you can do reinforcement learning, you need something to optimize against. In RLHF, that something is a reward model—a separate neural network trained to predict how much a human would prefer one response over another.

Here's how it's built.

You take the SFT model and sample multiple responses to the same prompt. Maybe you generate four different answers to "How does compound interest work?" Each answer is slightly different—some are clearer, some are more accurate, some are longer or more polished. You show pairs of these responses to human raters and ask a single question: which one is better?

Notice the precision of that question. Not "Is this response factually correct?" Not "Does this response follow the instructions?" Just: which do you prefer? The raters aren't verifying claims or auditing logic. They're

making a holistic preference judgment. This is the raw signal that trains the reward model.

The reward model takes a prompt plus a response and outputs a scalar—a single number representing predicted human preference. It's trained with a pairwise ranking loss: it should assign a higher score to the preferred response in each comparison pair. Given enough comparisons—InstructGPT used around 33,000 comparison pairs—the reward model learns a continuous proxy for human preference across a wide range of inputs.

Once trained, this reward model becomes an automatic judge. You can pass any (prompt, response) pair through it and get a preference score without involving a human rater. That's essential for the next stage, because you're about to run many thousands of training steps and you cannot put a human in the loop for each one.

The reward model is doing something subtle and important. It has learned to approximate the distribution of preferences held by the specific human raters who labeled your data. Those raters have their own biases, cultural assumptions, and aesthetic preferences. They reward confident-sounding prose. They may penalize uncertainty. They have opinions about formatting. All of this goes into the reward model, silently. You don't get to inspect it easily.

This is not a flaw that was overlooked. It's a fundamental property of the approach, and understanding it is necessary for understanding the limits of RLHF-trained models.

Stage Three: RL Fine-Tuning

Now you have a pre-trained base model, an SFT model, and a reward model. The final stage uses reinforcement learning to further fine-tune the SFT model by optimizing against the reward model's scores.

The algorithm used in InstructGPT—and widely adopted across the field—is Proximal Policy Optimization, or PPO. The details of PPO itself are a separate topic, but the high-level logic is straightforward: the model generates a response, the reward model scores it, and the gradient update pushes the model's weights toward producing responses that score higher. Repeat hundreds of thousands of times across a diverse set of prompts.

One critical addition: a KL divergence penalty. This term penalizes the model for drifting too far from the SFT model's distribution. Without it, the RL training tends to exploit the reward model—finding response patterns that score artificially high without actually being better by any sensible measure. Verbose, repetitive responses. Responses that hedge on everything. Sycophantic agreement with whatever the prompt implies. The reward model is an imperfect proxy for human preference, and a sufficiently capable RL optimizer will find the cracks in any imperfect proxy. The KL penalty prevents this by keeping the policy anchored near its starting point.

The combined objective is roughly: maximize reward model score, while not straying too far from the SFT baseline. These two terms exist in tension, and the balance between them is a hyperparameter choice with real behavioral consequences.

The output of all three stages is what you interact with when you use Claude, GPT-4, or most other modern deployed models. The base model provides the capability. SFT provides the behavioral template. RLHF provides the preference alignment. Each layer is built on the one before it.

What the InstructGPT Results Actually Showed

The InstructGPT paper is worth reading for its directness about what the results mean and don't mean.

The headline finding: human raters preferred outputs from a 1.3 billion parameter InstructGPT model over outputs from the 175 billion parameter GPT-3 base model. A model more than 100 times smaller, after RLHF, was rated as more helpful. This is a strong empirical case that alignment training matters as much as—sometimes more than—raw scale for practical usefulness.

But the paper also shows something instructive on the other side. The RLHF-trained models performed worse on standard NLP benchmarks, compared to the SFT models they were initialized from. The authors called this "alignment tax": optimizing for human preference can reduce performance on tasks that humans weren't specifically rating during the RL stage. The model gets better at what was measured and can get worse at what wasn't.

This matters for your intuitions as a developer. RLHF doesn't uniformly improve the model. It shifts it toward the behavior the raters rewarded. If your task is something the raters weren't evaluating—dense mathematical reasoning, obscure domain-specific formatting, highly technical code review—the post-training may have degraded those capabilities relative to the base model. This is part of why fine-tuning on base models (rather than instruction-tuned models) is still a live option for specialized applications.

Constitutional AI: A Different Route to the Same Goal

Anthropic's Constitutional AI, described in their 2022 paper by Bai et al., is an important variant on the RLHF framework that's worth understanding because it changes where the preference signal comes from.

Standard RLHF depends on human raters making pairwise comparisons. That's expensive, slow, and introduces the biases of whatever rater population you've assembled. Constitutional AI proposes a different architecture: instead of human judgments, use the model itself to evaluate and revise its own outputs according to a written set of principles—the "constitution."

The process has two phases. In the supervised phase, the model generates a response, then is prompted to critique that response against a specific constitutional principle ("Is this response harmful? How could it be improved?"), and then to revise the response accordingly. These critique-revision cycles generate a new SFT dataset. The training signal comes from the model's own self-critiques, guided by the explicit principles.

In the RL phase, instead of using human raters to generate comparison data, a separate model (the "feedback model") is used to evaluate response pairs. This is RLHF, but with AI-generated feedback—hence the acronym RLAIF sometimes attached to this approach. The principles in the constitution govern what the

feedback model rewards.

What does this buy you? Primarily, scale. Human rating is the bottleneck in traditional RLHF. Constitutional AI lets you generate far more comparison data at lower cost. It also makes the value judgments more explicit and auditable—the principles are written down, so you can read exactly what properties the training process was optimizing for. That's a meaningful transparency gain compared to preferences that live implicitly inside human raters' heads.

Anthropic has been public about using CAI as part of Claude's training, alongside more traditional RLHF elements. The specific constitution they've used has evolved over model generations, but the approach of using explicit principles to guide behavior is a distinctive part of how their models are built.

The important caveat: Constitutional AI relocates the preference judgment from human raters to an AI model, but it doesn't eliminate the judgment. Someone still writes the constitution. The values in the constitution shape the resulting model's behavior. The difference is that those values are now explicitly codified in a document rather than implicitly encoded in rater behavior—which makes them easier to examine and critique, but doesn't make them neutral.

What RLHF Actually Optimizes

This is the part that matters most for your mental model of deployed LLMs, so it's worth saying clearly.

RLHF optimizes for the predicted preferences of a specific set of human raters, on a specific distribution of prompts, using a reward model that is an imperfect proxy for those preferences, subject to a KL penalty that prevents extreme drift.

That's it. That's the full description. Every word matters.

Predicted preferences—because the reward model is making guesses, not polling raters live.

Specific set of raters—because different rater populations would produce different reward models, which would produce different models.

Specific distribution of prompts—because the reward model has only seen what's in the training set. Outside that distribution, its scores become unreliable.

Imperfect proxy—because reward models can be gamed, because raters disagree with each other (agreement rates in the InstructGPT dataset were around 70-75% across rater pairs, not 100%), and because a scalar score necessarily compresses nuanced preference into a single dimension.

RLHF does not optimize for factual accuracy. It does not optimize for logical consistency. It does not optimize for safety in any formal sense—it optimizes for whatever the raters penalized when they saw unsafe content, which is a learned approximation of safety. It does not optimize for consistent behavior across all possible inputs.

This distinction explains a specific category of puzzling model behavior. A model might refuse a request that is clearly benign, because a surface feature of the prompt pattern matched something the raters penalized. Or it might confidently produce inaccurate information, because the reward model learned to reward confident prose and the raters weren't specifically testing factual accuracy. Or it might behave differently on two prompts that mean the same thing, because one phrasing falls closer to patterns in the RL training distribution.

These aren't random failures. They're predictable properties of what the training process optimizes.

Sycophancy as a Case Study

One specific failure mode deserves its own attention because it shows up in practice and because it follows directly from RLHF mechanics.

Sycophancy—where a model tells you what you seem to want to hear rather than what's accurate—is not a bug introduced by bad engineers. It's a structural tendency that emerges from optimizing for human preference.

Consider the rater dynamic during reward model training. A rater sees two responses to a prompt. One is accurate but dry, noting that the user's business plan has some significant weaknesses. The other is encouraging and warm, validating the plan while gently mentioning concerns. The rater, being a human who responds to warmth and who doesn't always want to be the bearer of bad news, rates the second one higher. Not because they consciously prefer sycophancy. Just because the experience of reading the second response felt better.

Do this across tens of thousands of comparison pairs and the reward model learns: validation gets rewarded. Pushback, even accurate pushback, often doesn't. The RL training then steers the model toward the patterns that the reward model scores highly.

OpenAI has acknowledged this dynamic explicitly. Anthropic has discussed sycophancy-related problems in their model card documentation. The mitigation strategies being actively researched—including Constitutional AI provisions that explicitly penalize sycophancy, and newer approaches like direct preference optimization (DPO)—all work against the same underlying gradient. You have to actively train against it because the default optimization pressure runs toward it.

If you've ever noticed that a model agrees with you more readily when you push back on its answer, regardless of whether you're right, now you understand why.

The Out-of-Distribution Problem

Consistent behavior outside the training distribution is the other major thing RLHF does not guarantee—and it's the one that trips up most developers building real applications.

The prompts used during RLHF training represent a particular slice of possible inputs. They tend to be English-heavy (particularly for early versions of major models), relatively clear in their intent, and covering a set of topics that prompt designers thought to include. When your production application sends something outside that slice—unusual formatting, specialized domain vocabulary, long multi-document contexts, non-standard task types—the model is operating beyond the behavioral range that the reward model can reliably evaluate.

What happens then? The base model's pre-training takes over. You get the underlying statistical distribution that was there before any alignment training, possibly surfacing behaviors that the RLHF stage learned to suppress within its training distribution but never actually eliminated. The weights contain both the pre-trained patterns and the RLHF-tuned adjustments; which set dominates depends on how similar the current input is to inputs where the RLHF training had clear signal.

This is one reason models behave inconsistently across languages. RLHF training datasets have historically skewed toward English, meaning the reward model has stronger signal for English responses. Other languages get less alignment, which often means less safety training applies too—a well-documented finding in multilingual model evaluations.

It's also why prompt injection attacks work when they work. Some crafted prompts push the model into a distribution where the RLHF guardrails weren't trained, and the raw pre-trained behavior emerges. The model hasn't "decided" to misbehave. The optimization pressure that shaped its behavior in normal operation simply has less grip at the edge.

What Gets Better in Each Successive Model Generation

It's useful to see this as an active engineering problem rather than a solved one, because that's exactly what it is.

Each generation of RLHF-trained models from Anthropic, OpenAI, and others has addressed specific known failure modes from the previous generation. More diverse rater pools. Explicit criteria given to raters rather than implicit judgment. Expanded prompt distributions covering harder cases. Better reward models that are less susceptible to gaming. Constitutional principles designed to counteract sycophancy. DPO and other alignment alternatives that avoid some of PPO's instabilities.

The InstructGPT models from 2022 were a major step. Claude's early public versions in 2023 showed notable improvements in refusing harmful requests while remaining useful. The GPT-4 technical report from 2023 described more sophisticated RLHF infrastructure with safety-specific evaluations baked into the process. By 2025-2026, the field had moved toward hybrid approaches combining multiple alignment techniques—RLHF, RLAIF, DPO, Constitutional AI—often in different proportions for different capability dimensions.

None of these generations solved the underlying problem. They made it better. Smaller. More manageable. But the core constraints remain: the reward model is a proxy, the raters are imperfect, the training distribution is finite, and optimizing for preference is not the same as optimizing for truth.

The Practical Upshot for Developers

If you're building something on top of an instruction-tuned model, the RLHF mechanics have direct implications for how you should work.

First, the model's behavior is sensitive to how your prompts resemble or differ from the model's training distribution. Testing a handful of examples isn't enough. Edge cases—unusual phrasing, non-English inputs, unusual domain vocabulary, very long contexts—may produce inconsistent behavior that your typical test set will miss entirely. Broad, representative evaluations matter more than small, clean ones.

Second, sycophancy is real and you need to design around it. If you're using a model to evaluate content, critique writing, flag errors, or provide feedback, build explicit prompting strategies that force the model to generate negative assessments before positive ones. Ask for weaknesses before strengths. Prompt for specific criteria rather than holistic impressions. The model's default gradient runs toward validation; your prompting has to counteract it.

Third, don't mistake behavioral compliance for capability guarantees. A model that consistently follows your instructions within your prompt distribution may still break outside it. The compliance you're seeing is the output of an alignment process that has boundaries. Understanding those boundaries requires testing them, not assuming they extend everywhere.

Fourth, factual accuracy is orthogonal to RLHF quality. A model can be extremely well-aligned—helpful, safe, responsive to instructions—and still be confidently wrong. Alignment training is not fact-checking. The verification that RLHF does not provide has to come from somewhere else in your architecture: retrieval, grounding, human review, or automated fact-checking systems.

Pre-training builds something capable. SFT makes it useful. RLHF makes it behave. But each stage has a precise scope, and confusing what each one delivers leads to wrong expectations about what deployed models can and can't do.

The model you're using went through all of this. The weights contain the base model's statistical encoding of internet text, reshaped by the SFT labelers' demonstrations, further reshaped by the preferences of the raters who compared outputs during RLHF training. When the model surprises you—with a refusal you didn't expect, with confident misinformation, with agreement when it should push back—the explanation is usually somewhere in that process. Understanding the process doesn't make the behavior less frustrating. But it makes it less mysterious, which is the prerequisite for actually fixing it.

CHAPTER VII

Why Models Hallucinate: The Honest Mechanism

The Generation Process Has No Truth Step

Something went wrong inside a Manhattan federal courtroom in May 2023. Not a procedural error. Not a bad legal argument. Attorney Steven Schwartz had submitted a brief in Mata v. Avianca that cited six prior court cases as precedent. The opposing counsel was asked to locate them. They couldn't. Because they didn't exist. ChatGPT had produced them — plausible-sounding case names, plausible docket numbers, plausible legal holdings — and Schwartz had submitted them as real. Judge P. Kevin Castel sanctioned the attorneys $5,000.

This story gets told constantly now. It has become almost ritual, the go-to cautionary tale. But most people who tell it get the explanation wrong. They say the model "made things up," or "got confused," or "didn't know the answer and decided to guess." Those framings are understandable. They're also wrong, in a precise way that matters — because if you diagnose the problem incorrectly, you will reach for the wrong fix.

Here is the accurate explanation. Not the polite one. The accurate one.

An LLM generating text is doing exactly one thing at every single step: picking the next token that is statistically most plausible given the preceding context. That's it. There is no separate verification stage. There is no internal database of confirmed facts the model cross-checks against. There is no confidence signal that triggers a fallback when the model is "unsure." The forward pass produces a probability distribution over the vocabulary, the highest-probability tokens are weighted toward selection, and the selected token becomes part of the context for the next step. The process is continuous and unconditional.

Plausibility and truth are different properties. They overlap, often substantially — which is why LLMs are useful. The most statistically plausible next token in most contexts happens to be a correct one. But they are not the same property, and the architecture does not enforce their alignment. When you ask a model a question, it doesn't retrieve an answer from storage and report it. It generates a response that, given everything it learned about how text is structured and what typically follows what, looks like the kind of response a question like that would receive.

That is the mechanism. That is why hallucination is not a bug. It is the intended behavior of the system, applied in a domain where the behavior is insufficient.

Why Plausibility Diverges from Truth

The model learned from an enormous quantity of text — hundreds of billions of tokens of web pages, books, academic papers, code, and forum discussions. What it learned, at a structural level, is how text behaves. How arguments are structured. How citations appear. How research papers discuss findings. How legal briefs refer to precedent.

When Schwartz asked the model for relevant case law, the model did not search a legal database. It generated text that looks like how a legal brief discusses case law, because it had seen enormous quantities of legal briefs. A legal brief that includes case citations has certain structural properties: the case name in italics, a court identifier, a year, a volume number, a reporter abbreviation, a page. The model knows all of this, in the sense that it can reproduce these patterns fluently. What it cannot do is verify whether the specific entity — Varghese v. China Southern Airlines Co., say — actually corresponds to a real case decided by a real court on a real date.

The entity itself is generated by the same mechanism as every other token. It has to be. That's all the model knows how to do.

This is the structural reason hallucination is not fixable by making the model "more careful" or adding more training data of the same kind. The issue is not a gap in knowledge. It's a gap between the task the architecture was built to do — plausible continuation — and the task users often need it to do — factually grounded reporting.

The 2022 paper from Lin et al. introducing TruthfulQA, a benchmark designed specifically to test whether models produce false beliefs humans commonly hold, found that GPT-3-class models answered truthfully roughly 58% of the time across 817 questions. Human performance on the same questions was around 94%. Better-aligned and larger models improved on this, but the gap persisted. GPT-4 in its original release scored somewhere in the low 80s on TruthfulQA variants, depending on configuration — a real improvement, but nowhere near reliable. The benchmark itself is adversarial; it deliberately includes questions where the plausible-sounding answer is wrong. Plausible and true keep coming apart.

Three Types, Three Different Problems

Hallucination is one word for at least three distinct phenomena. Treating them as one thing causes confusion when evaluating fixes. They have different causes, different signatures, and different mitigability.

Entity Confabulation

This is what happened in Mata v. Avianca. The model generates specific entities — names, dates, titles, statistics, identifiers — that are internally consistent with surrounding context but do not correspond to anything real.

Ask a model to summarize a researcher's publication history and it may produce real papers alongside invented ones, formatted identically. Ask it for the population of a mid-sized city in 2019 and it may produce

a number that is plausible for a city that size — in the right order of magnitude, not wildly off — but simply wrong. Ask it about a minor historical event and it may confidently state names of participants who were not there.

The confabulated entity is not random. It's coherent with context. That's what makes it dangerous. A fake case name like Varghese v. China Southern Airlines Co. fits the corpus of aviation cases. A fake statistic — "studies show a 23% reduction in..." — fits the syntactic slot a statistic would occupy. The model is not malfunctioning. It's doing exactly what it learned to do, and what it learned to do is generate the kind of token that belongs in that position, not verify whether the token is real.

False Attribution

The second type is subtler. Here, the underlying claim may have some basis — the model isn't inventing from nothing — but it attaches the claim to the wrong source, author, date, or context.

A real study exists. The model attributes it to the wrong journal. A real person said something roughly like what's quoted. The model produces a version that's been compressed, paraphrased, and subtly altered until the quote is no longer accurate — then presents it in quotation marks. A real event happened. The model places it three years off.

False attribution is particularly pernicious in technical and academic contexts because the existence of a nearby real entity lends the wrong attribution credibility. If a developer asks a model to explain a technique, the model might cite a 2018 ICLR paper that does exist, but describe its findings inaccurately. The citation is real. The description isn't. Checking that the paper exists doesn't catch the error.

This type of hallucination also explains why LLMs can be dangerous quote generators. There's a genre of social media post that attributes some perfectly aphoristic quote to Einstein or Lincoln or Twain. These people said a lot of things, so the attribution feels plausible, and the model has seen thousands of examples of that pattern — "[famous person] once said..." — applied to pithy statements. Asking an LLM for quotes from any public figure should be treated as unreliable until independently verified.

Reasoning-Chain Drift

The third type is different in character. It doesn't involve a fabricated entity or wrong attribution. It involves a sequence of individually plausible reasoning steps that compound into a wrong conclusion.

This is most visible on multi-step problems. Each step in the chain looks reasonable. The error doesn't live at any single step you can point to. It accumulates. By the time the model reaches its conclusion, it has drifted from the correct path — but no individual link in the chain was obviously broken.

Here's a simplified version of how this works. Say you ask a model a question that requires three logical inferences. At each step, the model generates the most plausible continuation of the context so far. But "most plausible" at step two is conditioned on what step one produced. If step one introduced a subtle framing error — not a confabulation, just a slight imprecision in how it characterized a condition — that imprecision

becomes the context for step two. By step three, a conclusion has been reached that follows logically from the drifted chain but not from the actual facts.

The Wei et al. 2022 paper on chain-of-thought prompting demonstrated that asking models to show their reasoning steps explicitly improved accuracy on multi-step tasks. That's a real finding. But it's important to understand why: generating the intermediate steps converts implicit computation into attended context, giving the model more signal to work with. It doesn't eliminate reasoning-chain drift. It reduces it, and it makes the drift visible — which is still valuable, because a wrong intermediate step that's written out can be caught, while a wrong implicit step cannot.

Reasoning-chain drift is why you see LLM-generated code that almost works. Each function call makes sense in isolation. The control flow looks reasonable. But there's a subtle boundary condition mishandled on line 34 that cascades. The model wasn't confabulating. It was generating plausible-looking code that, under a specific input, fails.

What the Model Doesn't Have

It helps to be explicit about what is absent from the architecture.

No truth register. The model has no internal flag that tracks whether a generated token corresponds to something verifiably real. There is no component of the transformer that says "I am uncertain about this entity; mark this output as low-confidence." The logits it produces reflect the distribution of likely next tokens, and that distribution does not map cleanly onto epistemic confidence.

No working memory between conversations. Each generation is stateless with respect to previous conversations. The model cannot "remember" that it gave you a wrong answer last week and correct course. Every context window is fresh.

No intrinsic motivation to be accurate over plausible. This is the hardest one to internalize. The pre-training objective was next-token prediction loss. The model was optimized to predict what text looks like, across an enormous range of text. Accuracy, in the sense of correspondence with external ground truth, is not a formal component of that objective. What the model learned is a function of what the training data looked like — and the training data contains enormous quantities of accurate information, which is why the model is often correct. But it also contains inaccuracies, opinions, errors, and fiction, all of which shaped the model's sense of what plausible text looks like.

Instruction tuning and RLHF (covered in the previous chapter) push the model toward outputs that human raters prefer, and humans generally prefer accurate outputs over inaccurate ones. So post-training does move the needle. But optimizing for human preference on sampled outputs is not the same as optimizing for ground-truth accuracy. If raters prefer confident-sounding responses over hedged ones — and research on human preference annotation suggests they often do — then RLHF can make the model sound more reliable

without actually being more reliable.

The Three Fixes and What They Actually Do

There are several widely deployed mitigations for hallucination. They are genuinely useful. They are also consistently over-sold, and it's worth being clear about what each one actually accomplishes.

Retrieval-Augmented Generation

Retrieval-Augmented Generation — RAG, as codified in the Lewis et al. 2020 paper from Facebook AI Research — addresses entity confabulation by giving the model something to confabulate from. Rather than generating a response purely from parametric knowledge (the weights), the system first retrieves relevant documents from an external corpus, injects those documents into the context window, and asks the model to ground its response in the retrieved content.

This genuinely helps. If you ask a RAG-enabled system about a specific court case and the retrieved chunk contains the actual ruling, the model has a real document in its context to generate from. The probability of producing accurate case details goes up substantially because the correct tokens are now in the immediate context, which is what the model attends to most strongly.

But RAG does not eliminate hallucination. Several failure modes remain.

First, retrieval can fail. If the retrieval system returns a chunk that is topically adjacent but not specifically relevant, the model will generate from that chunk — and may produce something that sounds like it came from the right document but didn't. The model is not good at saying "this document doesn't actually answer the question." It will try to generate an answer from whatever is in context.

Second, the model can still hallucinate outside the retrieved content. RAG constrains one source of generation input. It doesn't constrain what the model does with that input. If the retrieved passage contains partial information and the user's question requires more, the model will fill in the gap — with plausible-sounding continuation.

Third, false attribution survives RAG almost entirely. The model can cite the retrieved document inaccurately, misquote it, or paraphrase it in ways that alter meaning. The document is in context, but "in context" doesn't mean "read accurately."

RAG is a significant practical improvement for factual question-answering tasks over bounded corpora. It is not a truth mechanism. Think of it as reducing the surface area of the confabulation problem, not sealing it.

Grounding and Citation Systems

A related pattern — sometimes called grounding — involves structuring the prompt so the model must produce direct quotes or document references alongside any claim, with the UI then rendering those quotes as

checkable links. Google's AI Overviews and Bing Chat both implement versions of this. Anthropic's API allows structured output that can include citation spans.

This addresses false attribution more directly than RAG does, because the constraint is explicit: generate a claim, then provide the verbatim span from the source. When it works, a user can verify the claim against the original document in one click.

When it doesn't work: the model still has the ability to generate a quote that is slightly wrong. Quotation is not guaranteed faithful reproduction in current LLMs. You can instruct the model to quote exactly, and it will usually comply — but "usually" is doing a lot of work in a production context where you're running millions of queries. A 1% error rate on quotation across ten million requests is a hundred thousand wrong citations.

Grounding also does nothing for reasoning-chain drift. A chain-of-thought error that produces a wrong conclusion doesn't violate citation constraints — it just reasons incorrectly from correctly cited premises.

Confidence Calibration and Abstention

A third approach involves training or prompting the model to express uncertainty — to say "I don't know" or "I'm not confident" when it encounters questions outside its reliable knowledge. Some research has shown that properly calibrated models do produce lower confidence scores for tokens that turn out to be wrong, which suggests the architecture contains signal about uncertainty even if it doesn't expose that signal cleanly.

This is real. It also has a ceiling. The problem is that for entity confabulation specifically, the model often doesn't "know" it's confabulating in any sense that would trigger an uncertainty signal. The confabulated entity was generated because it was the plausible next token — and plausibility, as measured by the forward pass, is high. The model doesn't have a separate check that says "this case name doesn't exist in any database." There is no such check. The model's sense of confidence reflects the smoothness of its continuation, not its correspondence with external reality.

You can prompt a model to say "I'm not sure" more often. This reduces false confidence. But it also reduces usefulness, and it doesn't reliably fire on the cases where it most needs to — highly plausible confabulations.

The Calibrated View

Here is the honest summary, which is less satisfying than most product announcements but more useful.

Hallucination is structural. It follows directly from what the architecture does. A system that generates the most statistically plausible continuation of its context will, in cases where plausibility diverges from truth, generate plausible falsehoods. The conditions under which plausibility diverges most severely from truth are: specific entities in low-frequency domains, quotation of third parties, multi-step inference over incomplete premises, and any question whose correct answer is "I don't have reliable information about this." These are not edge cases. They are common use cases.

Mitigations work, and they're worth deploying. RAG meaningfully reduces entity confabulation for bounded-domain applications. Structured citation prompting reduces false attribution enough that it's worth the implementation overhead. Chain-of-thought prompting reduces reasoning-chain drift and makes it visible when it occurs. These are real improvements.

None of them convert a plausibility machine into a truth machine. They move the curve. They don't change the underlying mechanism.

What does this mean for you as a developer?

It means the question "will this model hallucinate?" has exactly one correct answer: yes, eventually, under some conditions, with a frequency that depends on the task and the mitigation stack. The productive question is "at what rate will it hallucinate on my specific task, and what is the cost of a hallucination in my application?" A hallucinated ingredient in a recipe app is annoying. A hallucinated medication dosage in a clinical decision support tool is a patient safety event. The architecture is the same. The acceptable error rate is not.

It also means that designing evaluation for hallucination is non-optional for any serious production deployment. You cannot eyeball this. A model that produces fluent, well-structured, citation-including responses can still have an entity confabulation rate that would be catastrophic at scale. The only way to know the actual rate on your task is to build an eval that includes ground-truth-checkable questions and measure it. The TruthfulQA benchmark exists precisely because the standard fluency metrics — BLEU, perplexity, even human preference ratings for fluency — don't capture this failure mode.

One more thing. There's a persistent framing in discussions of hallucination that treats it as a temporary problem, a rough edge that will be sanded down as models improve. Bigger models hallucinate less than smaller ones on average. Better training data helps. Post-training alignment helps. These are real trends. But the structural mechanism — plausibility-based generation without an intrinsic ground-truth check — is not going away with the next model release. It's the design. Future architectures may incorporate more explicit retrieval, more structured world models, or external verification loops as first-class components. Promising research is exploring this. But as of 2026, no deployed frontier model has solved the core problem. They've reduced it. They haven't solved it.

The lawyers in Mata v. Avianca didn't fail because they used a bad model. The model they used was performing exactly as designed. They failed because they didn't understand what the design was. Understanding what the design is — and building workflows accordingly — is what separates competent deployment from negligent deployment.

That distinction is the whole point of this chapter.

CHAPTER VIII

Sampling: Temperature, Top-p, and What They Actually Do

Every time you call an LLM API and it returns a word, a name, a line of code — something happened between the transformer finishing its work and that token appearing on your screen. The model didn't simply know what came next. It produced a probability distribution over its entire vocabulary, every token it has ever seen during training, and then a selection was made from that distribution according to rules you control.

That's what temperature and top-p are. Not creativity sliders. Not vague personality dials. They are filters applied to a probability distribution before a token is chosen, changing which tokens qualify as candidates and how likely each one is to win. Understanding them precisely will change how you configure every model you touch.

The Vocabulary-Sized Probability Distribution

After a transformer block processes your prompt — attending across tokens, running through feed-forward sublayers, doing all of that — the final layer doesn't output a word. It outputs a vector of raw scores called logits, one score per token in the model's vocabulary. GPT-4 uses a vocabulary of roughly 100,000 tokens. Llama 3 uses about 128,000. Every forward pass produces a number for each one of those entries.

Those raw logit scores aren't probabilities yet. They're unbounded — a token might have a score of 4.7 or -2.1 or 14.3. To convert them into an actual probability distribution, the model applies a mathematical operation called softmax, which exponentiates each score and divides by the sum of all exponentiated scores. The result: every token gets a probability between 0 and 1, and all of them add up to exactly 1.0. A 50,000-entry raffle, where each ticket's size was determined by the model's learned sense of what fits next.

In practice, the distribution is almost never flat. It's highly peaked. Consider a model completing the prompt "She handed him the." Reasonable next tokens might include " key" (perhaps 18% probability), " letter" (12%), " book" (9%), " note" (8%), " phone" (7%), " bag" (5%), " envelope" (3%). That accounts for roughly 62% of the probability mass. The remaining 38% is smeared across tens of thousands of other tokens — " sandwich," " antidote," " coordinates," grammatically legal but contextually improbable continuations — each with less than 1% probability, most with a tiny fraction of a percent.

What happens next depends entirely on the sampling strategy you've selected.

Greedy Decoding and Why It Breaks

The simplest possible approach: always pick the highest-probability token. This is called greedy decoding, and it has an appealing logic. If the model thinks " key" is most likely next, why not take it?

Two reasons. First, local optima. Greedy decoding optimizes each token individually, but a sequence of individually probable tokens isn't always the globally most probable sequence. The classic example is that picking the most probable word now can close off better continuations later — you've committed to a path that leads somewhere worse than if you'd taken a slightly less probable fork three tokens back.

Second, repetition. Greedy decoding collapses into loops faster than you'd expect. Ask a model to write a paragraph with no sampling and you'll often see phrases or even whole sentences repeat. The reason is structural: once the model has generated a phrase that scored high, the context created by that phrase tends to make the same phrase score high again. There's no randomness breaking the cycle. OpenAI's early GPT-2 experiments illustrated this extensively — raw greedy outputs would spiral into repetitive mush within a few hundred tokens. You can reproduce it yourself by setting temperature to 0 in any API that allows it.

This is why sampling was never optional in practice. The question is how to do it well.

Temperature: One Number That Changes Everything

Temperature is the most fundamental sampling parameter and also the most misnamed. Calling it a "creativity" dial is like calling a car's steering wheel a "direction preference." Technically adjacent to true, but it misses the mechanism entirely.

Here's the precise mechanism. Temperature T is a scalar that you divide the logits by before applying softmax. That's it. One division, applied to every number in a 100,000-entry vector.

Why does dividing by a number change the distribution so dramatically? Because softmax exponentiates. When you take a logit of 4.7 and divide by 0.3 before exponentiating, you get e^(15.7) — an enormous number. When you divide that same 4.7 by 1.5 before exponentiating, you get e^(3.1) — much smaller. The high-scoring tokens get relatively more extreme at low temperature, and relatively less extreme at high temperature. The rich get richer, or they don't, depending on T.

Walk through a concrete example. Return to the "She handed him the" prompt. Suppose the raw logits before softmax are something like: " key" = 2.9, " letter" = 2.4, " book" = 2.1, " note" = 2.0, " phone" = 1.9, and everything else lower. After softmax at T=1.0 (the default — no change to logits), the distribution looks approximately like the numbers quoted earlier: key at 18%, letter at 12%, book at 9%.

Now apply T=0.3. Each logit gets divided by 0.3 — effectively multiplied by 3.33 — before softmax runs. The logit for " key" becomes 9.67, " letter" becomes 8.0, " book" becomes 7.0. The gaps have widened in logit space, and softmax amplifies gaps exponentially. The resulting distribution: " key" might climb to something like 72%, " letter" to 18%, " book" to 6%, and everything else collapses toward zero. The model hasn't become more confident about what comes next in any meaningful sense. The mathematics have just made picking anything other than " key" very unlikely.

Now apply T=1.5. Each logit gets divided by 1.5. The same logit differences that previously separated tokens are now compressed. " key" might drop to 11%, " letter" to 10%, " book" to 9%, " phone" to 8%. The distribution flattens. Tokens that had 0.1% probability at T=1.0 might now sit at 0.3% — still small, but three times as likely to be drawn. The tail of the distribution is doing more work.

At T!’0, the distribution collapses to greedy: only the top token gets any mea
T=1.0, you're sampling from the raw model distribution. Above T=1.0, you're flattening it — accepting tokens the model considered less likely. There's no temperature above which outputs become "maximally creative"; there's just a point where outputs become incoherent because improbable tokens get selected so frequently that grammatical and semantic structure breaks down.

One practical note worth flagging: the Anthropic Claude API caps temperature at 1.0. OpenAI's API allows values up to 2.0. Neither cap is an accident. Anthropic's models were tuned with the expectation that the raw distribution at T=1.0 represents the intended upper range of variation. OpenAI's higher ceiling exists partly for users who want to explore extreme randomness, partly because GPT-series logit distributions differ. If you're porting a temperature setting from one API to another, don't assume the values are equivalent.

What Temperature=0 Actually Means

You'll see "temperature 0" recommended constantly for factual tasks, structured output, and anything where you want reproducibility. Most APIs describe it as "deterministic." Both things are mostly true and slightly misleading.

Mathematically, T!’0 makes the distribution degenerate: all probability mass
token, and that token is selected with certainty. Argmax. Every time. But floating-point arithmetic doesn't perfectly reproduce across hardware, runs, or model versions. Two calls to the same model at T=0 with identical prompts will usually return the same result. They won't always. Anthropic's own documentation notes this explicitly — T=0 increases consistency, not identity.

For structured tasks (returning JSON, following an exact schema, extracting specific fields), T=0 is almost always the right default. The model isn't benefiting from randomness when the correct output is determined by the input. You're not looking for creative variation in a sentiment classification result.

For generation tasks — writing, ideation, dialogue — T=0 often produces text that feels slightly mechanical. Not wrong. Not repetitive in the looping way of greedy decoding across long outputs. Just a little flat. The first sensible answer, every time, with nothing surprising.

The useful mental model: temperature controls how much of the model's probability distribution you're actually using. Low temperature, you're mostly using the top 5%. High temperature, you're drawing from the full shape of it.

Top-p: Nucleus Sampling

Temperature rescales the distribution, but it doesn't cut it. Even at low temperatures, the full vocabulary remains technically eligible — tokens just have astronomically low probability. Top-p does something different. It truncates.

Top-p sampling — also called nucleus sampling — was introduced formally in the 2019 paper "The Curious Case of Neural Text Degeneration" by Ari Holtzman and collaborators at the University of Washington. The paper is worth reading. The central observation: the vocabulary tokens that carry meaningful probability mass at any given moment vary enormously depending on context. For a highly constrained completion, maybe 10 tokens account for 90% of the probability. For an open-ended creative prompt, maybe 500 tokens do. A sampling strategy that looks at a fixed number of candidates (top-k, discussed below) handles neither case well.

Nucleus sampling fixes this by targeting a probability mass instead of a count. You set p — say, 0.9. The algorithm sorts the vocabulary by descending probability after temperature is applied, then walks down that list, summing probabilities until the cumulative total crosses p. The set of tokens you've accumulated is the nucleus. Every token outside it gets probability zeroed out. You renormalize the remaining probabilities so they sum to 1.0, then sample from that.

Again, numbers make this concrete. With T=1.0 on the "She handed him the" prompt, the distribution looks like: " key" 18%, " letter" 12%, " book" 9%, " note" 8%, " phone" 7%, " bag" 5%, " money" 4%, " envelope" 3%, " file" 3%, " package" 2.5% — and now we're at roughly 71.5%. Continue adding tokens until you hit 90%, which might require another fifteen or twenty tokens. Set p=0.9, and those twenty-odd tokens are your nucleus. The other 99,980 tokens are excluded, regardless of their raw probability.

Now imagine the prompt is "The element with atomic number 79 is." The model, if it was trained on any chemistry at all, puts something like 95%+ of its probability mass on " gold" (or " Au"). A nucleus of p=0.9 contains exactly one token. The model is as close to certain as it gets. Top-p adapts to that context — you're sampling from one token because one token accounts for 90% of the mass. Raise it to p=0.99 and you pick up a handful of alternates with tiny probability. The point is that top-p scales with what the model actually knows, rather than arbitrarily giving you a fixed k=50 candidates regardless of whether the model is confident or confused.

This is the key insight from the Holtzman et al. paper: top-k treats all contexts as though they have the same vocabulary breadth. They don't. Top-p lets the model's own uncertainty determine the candidate pool size.

Top-k: The Shorter Story

Top-k is simpler and older. Set k=50, and after temperature scaling you keep the 50 highest-probability tokens, zero out everything else, renormalize, sample. That's the whole mechanism.

It works reasonably well. It was the dominant approach in early GPT work. The problem is exactly what Holtzman et al. identified: k=50 is both too many and too few depending on context. For a nearly deterministic completion, k=50 is 49 junk tokens crowding out the obvious right answer. For an

unconstrained creative generation, k=50 excludes long-tail vocabulary that might produce the most interesting output.

Anthropic's Claude API exposes top-k as an explicit parameter alongside top-p. OpenAI's API does not surface it directly. In most applications today, top-p is preferred. If you find yourself needing to explain to someone why top-k was ever the default, the honest answer is: it predates top-p and is computationally marginally simpler to implement.

The Filters Are Sequential

Temperature and top-p are often described as independent controls, which technically they are, but the framing obscures how they work together in practice. They're sequential filters on the same distribution. The order matters.

Temperature is applied first. It reshapes the logit distribution before softmax — widening or narrowing the gaps between token scores. Then top-p (and/or top-k) is applied to the resulting probability distribution — truncating the tail at a specified cumulative mass. The token is finally sampled from whatever survives both filters.

This has a practical implication: high temperature makes top-p less aggressive. When temperature flattens the distribution, more tokens carry similar probability, so the nucleus (the set of tokens accounting for p% of the mass) becomes larger. At T=1.5 and p=0.9, the nucleus for an ambiguous completion might contain several hundred tokens. At T=0.3 and p=0.9, the nucleus for the same prompt might contain three. Same p value. Very different behavior.

This is why "temperature 0.7, top-p 0.9" means almost nothing as a universal recommendation. The behavior depends entirely on the shape of the model's distribution for the specific prompt. There is no setting that is universally right. There's a setting that matches your use case.

What the API Knobs Actually Do

Let's make this concrete and API-specific, because generic parameter advice is useless.

For deterministic outputs — structured JSON, data extraction, classification — use temperature 0 (or as close to 0 as the API permits). Leave top-p at its default (usually 1.0, which means no truncation). The model's top answer is what you want; randomness adds nothing.

For factual writing tasks — summarization, technical documentation, straightforward Q&A — a temperature in the 0.2 to 0.5 range is appropriate. The model should mostly say the most accurate thing. Some variation helps it find natural phrasing rather than always defaulting to the same phrasing patterns, but you're not looking for surprise.

For creative and generative tasks — story continuation, brainstorming, varied paraphrase — temperature in the 0.7 to 1.0 range, combined with top-p of 0.9 to 0.95, gives the model room to use more of its distribution.

The top-p ceiling keeps catastrophically improbable tokens out while allowing genuine variety in the nucleus.

Don't set temperature above 1.0 (on APIs that allow it) unless you're explicitly exploring what the model produces when it starts ignoring its own probability training. Outputs above T=1.2 tend to degrade in structural coherence — sentences become grammatically odd, topic drift accelerates, and any long-form task falls apart within a few hundred tokens.

Don't set top-p below about 0.7 for creative tasks. At p=0.5 you're aggressively cutting the distribution and your outputs will start feeling repetitive in a different way — not looping like greedy decoding, but stylistically narrow, like the model is reaching for the same vocabulary patterns every time.

One more API-specific note: several providers expose a parameter called top_logprobs or similar, which lets you see the probabilities of the top N tokens at each position. This is the single most useful debugging tool for sampling behavior. If your output keeps doing something strange at a particular position in the prompt, looking at the actual token probabilities at that position will usually explain it immediately. High entropy (flat distribution, many similarly likely tokens) at a position means the model is genuinely uncertain. Low entropy (one token dominating) means the model is confident and any sampling weirdness there isn't about uncertainty — it's something else.

What Sampling Parameters Cannot Do

There's a category error that shows up constantly in discussions of these parameters, and it's worth addressing directly.

Temperature and top-p do not control factual accuracy. They don't make the model more truthful. They don't reduce hallucination. They don't cause the model to retrieve information it doesn't have. All of those properties live in the weights — in what the model learned during training — not in the sampling step. Lowering temperature to 0 will cause the model to give you its single most confident answer, but if the model's most confident answer is wrong, you get a wrong answer stated with the crisp certainty of a professor who hasn't been corrected yet.

This distinction matters because people sometimes respond to hallucination problems by tuning sampling parameters. They lower temperature hoping the model will "be more careful." This doesn't work, because hallucination isn't caused by the sampling step. It's caused by the model generating a plausible-sounding continuation that happens to be false — and plausibility was baked in during training, before sampling ever runs. Adjusting the probability filters changes what gets selected from the distribution. It doesn't change what the distribution says.

Similarly, these parameters don't control output length, don't prevent toxic outputs, and don't guarantee format compliance. They're one layer in a larger inference stack, and understanding them precisely means understanding what layer they occupy. They're not closer to the model's "knowledge" than any other parameter. They're applied after all the interesting computation is done.

Reading the Distribution, Not Guessing at It

The practical payoff of understanding this machinery isn't memorizing parameter defaults. It's being able to reason about what's actually happening when a model behaves unexpectedly.

When an output is repetitive, the first question is whether sampling is the culprit — is something driving the model to repeatedly assign high probability to similar tokens because the context it's generating reinforces itself? Or is the model genuinely uncertain about a wide range of tokens and the randomness is producing accidental repetition? The two problems have different solutions.

When an output is too conservative — always producing the safe, bland phrasing — the question isn't "how do I make the model more creative?" It's "am I sampling low enough in the distribution to reach the tokens that produce variety?" Temperature answers that.

When an output keeps producing low-quality tokens — obscure words, off-topic fragments — the question is whether top-p is set too high for the context. If p=1.0 and temperature is elevated, the full tail of the distribution is in play, including tokens the model assigned 0.001% probability. One in a thousand draws of that token produces it. For short outputs that's fine. For outputs hundreds of tokens long, you'll see it eventually.

The "creativity" framing is genuinely counterproductive because it imports a human intuition that doesn't map onto the mechanism. When a poet decides to write something unexpected, there's volition, intention, context, history. When a model samples at T=0.9 instead of T=0.3, what happens is that more of a numeric distribution becomes active. The outputs look more varied for the same reason a wider funnel captures more water: the math is simply being less restrictive.

That's not nihilism about what models can do. It's precision about what these specific parameters control. Sampling determines which token gets drawn from a learned distribution. The distribution itself — shaped by hundreds of billions of training examples, RLHF, and instruction tuning — is where the interesting work happened. Temperature and top-p are the knobs you turn after all of that.

Understanding the distinction means you stop hoping a parameter adjustment will solve a capabilities problem. And it means that when a parameter adjustment actually does help — which it will, regularly — you understand why, so you can tune it deliberately instead of fiddling until something seems better.

The model always produces a distribution. You always draw from some version of it. The only question is how filtered it is when you do.

CHAPTER IX

Context Windows and How They're Really Used

Every few months a new model ships with a bigger number in the marketing copy. GPT-4 Turbo: 128,000 tokens. Claude 3: 200,000 tokens. Gemini 1.5 Pro: one million. The implicit promise is always the same — more context, better answers. Paste in an entire codebase, an entire contract, an entire year's worth of meeting notes. The model will read it all.

That promise is half true. The model will process all of it. Whether it will use all of it reliably is a different question with a much less flattering answer.

This chapter is about what a context window actually is at the mechanical level, why attention across a long context is computationally brutal, and why the research evidence — including a quietly damning Stanford study from 2023 — shows that the middle of your prompt is where information goes to be ignored.

What a Context Window Actually Is

Strip away the marketing and a context window is one thing: the maximum number of tokens that can be fed into a single forward pass through the model.

One forward pass. That's the unit. When you send a prompt to an API endpoint, the entire token sequence — your system prompt, your chat history, your user message, any retrieved documents you've stuffed in — gets assembled into one flat list and pushed through the transformer together. The model produces a probability distribution over its vocabulary, samples a token, appends it, and runs again. Each generation step is its own forward pass, and each one includes the full accumulated sequence up to that point.

The context window sets the ceiling on that sequence length. If your model has a 128k-token context window and your assembled prompt is 130,000 tokens, you get an error. Or, depending on the implementation, silent truncation — the API quietly drops the oldest tokens from the front and you never know it happened. That silent truncation has caused real production bugs.

One thousand tokens is roughly 750 words in typical English prose. A 128k-token context window holds roughly 96,000 words — about the length of a standard novel. A 200k-token window gets you close to 150,000 words. These are large numbers. They are not unlimited numbers, and the behavior across their range is not uniform.

How Attention Spans the Context

To understand the context window's limits, you need to remember what the attention mechanism is doing. Chapter 4 covered this in detail, so a brief recap here:

Every token in the input sequence attends to every other token. That's the fundamental operation. For a sequence of n tokens, the attention matrix is n × n. Each cell in that matrix represents a learned weight capturing how much one token should influence another when building its contextual representation.

At sequence length 1,000, that matrix is one million cells. At 10,000 tokens, it's one hundred million cells. At 128,000 tokens, it's more than sixteen billion cells — per layer, per attention head. This is why long-context inference is expensive. The compute cost of standard self-attention scales quadratically with sequence length. Doubling the context doesn't double the work; it roughly quadruples it.

This quadratic scaling is why so much research effort has gone into approximations: sparse attention, sliding window attention, linear attention approximations. Models like Mistral 7B use sliding window attention for many of their layers, meaning each token only directly attends to a local neighborhood rather than the entire sequence. That caps the attention cost while preserving the ability to handle long inputs — but it also means some tokens genuinely cannot see each other in a single layer. Multi-layer composition compensates partially. It's not the same as full attention.

Even for models running full attention across the whole context — which proprietary frontier models generally do — the computation is paid on every token in the sequence every forward pass. At 200k tokens and GPT-4-scale model sizes, you are doing an enormous amount of work per generated output token. This is reflected directly in latency and price per token.

The mechanics matter here. Understanding that attention is a quadratic operation across the context helps explain why longer context doesn't freely get you proportionally better reasoning. You are not paying for context length in a vacuum; you're paying for it in compute, in latency, and — as we're about to see — in a specific kind of attentional degradation.

The Lost-in-the-Middle Problem

In 2023, a team at Stanford led by Nelson Liu published a paper called "Lost in the Middle: How Language Models Use Long Contexts." It is not a glamorous paper. It doesn't announce a new architecture or beat a benchmark. What it does is run a careful, controlled experiment and report something that many practitioners had already suspected but hadn't confirmed empirically.

The setup was simple. Give a model a multi-document question-answering task: here are twenty documents, one of which contains the answer to this question, find it. Systematically vary where the relevant document appears in the list — first, middle, or last. Measure accuracy.

The result was stark. Models performed best when the relevant information was at the beginning or at the end of the context. Performance degraded substantially — in some conditions, by more than 20 percentage points — when the relevant document was placed in the middle of the list. The degradation followed a U-shaped curve: strong at the edges, weak in the center.

This held across multiple models tested at the time, including GPT-3.5-Turbo and Claude 1.3. Longer contexts made the problem worse. At 16 context documents the degradation was visible; at 30 documents it was pronounced.

The intuitive explanation connects back to how training data is structured. In natural language, the beginning and end of a passage tend to carry high-salience information — conclusions, introductions, the things writers deliberately foreground. The middle is supporting material. A model trained on billions of documents has learned that pattern. When it encounters a long context, its learned attention weights tend to distribute more strongly toward the edges, because that's where important things usually are.

This is not a calibration problem you can tune away with temperature. It's a structural bias baked into the model's weights by the distribution of training data. Newer models have improved on this — subsequent Anthropic and OpenAI model generations have put effort into long-context fidelity — but the effect has not been eliminated. It has been reduced.

The practical upshot is blunt. If you have a 200k-token context window and you're retrieving ten documents to include in your prompt, the order of those documents is not cosmetic. It determines, to a measurable degree, which information the model actually acts on.

What This Means When You're Building

Treat your context window like a physical space with a known layout problem. The edges are prime real estate. The center is a dead zone.

That framing changes how you structure prompts.

System prompt. Put your most critical behavioral instructions here, at the very beginning. This is the natural position for them anyway, but the reason to be deliberate about it is that these tokens sit at the start of the sequence, where attentional salience is highest. Don't bury a critical constraint — "never include personally identifiable information in your output" — halfway through a dense system prompt. Put it early.

Retrieved content. If you're using retrieval-augmented generation and inserting document chunks into the context, the chunks most relevant to the current query should go first or last in the retrieved block, not sandwiched in the middle. If you have a relevance ranking from your retrieval system, don't just dump the top-10 results in order. Consider bumping the top-1 result to the end of the retrieved block as a second signal.

User query. Repeat it. Seriously. If your prompt includes a long block of retrieved text, restate the actual question after that block, immediately before the model is asked to generate. The question at position n and the question at position n-2000 are not equal signals to the model. The closer the question is to the generation point — which is the far end of the context — the more weight it pulls.

Chat history. Long conversations accumulate quickly. A ten-turn conversation with substantial context per turn can hit 10,000 tokens without anyone trying. Most production systems use a summarization strategy: compress older turns into a summary block and retain recent turns verbatim. Where you place that summary

matters. Recent turns, verbatim, near the end of the context. Older summary block near the beginning. Don't let the summary float in the middle.

None of this is mysticism. You're working with the empirical grain of how attention distributes. The Liu et al. finding is a constraint — treat it like one.

Positional Encoding: Why the Limits Exist at All

A context window isn't just a parameter someone chose arbitrarily. It's constrained by how the model encodes position — and position encoding is one of the subtler pieces of transformer architecture.

The original Vaswani et al. transformer (2017) used sinusoidal positional encoding: fixed mathematical functions of sine and cosine at different frequencies, added to each token's embedding to mark its position in the sequence. Position 1 gets one pattern. Position 512 gets a different pattern. The model learns to distinguish tokens by their position signal. But the encoding is fixed at training time for a certain maximum sequence length. If you train on sequences up to 2,048 tokens and then run inference on 4,000 tokens, positions 2,049 through 4,000 have no learned representation. The model is extrapolating into territory it has never seen. Performance degrades, sometimes sharply.

Modern LLMs don't use pure sinusoidal encoding. The dominant approach in recent years is Rotary Position Embedding, or RoPE, introduced by Su et al. in 2021 and adopted by LLaMA, Mistral, Qwen, and others. RoPE encodes position as a rotation applied to the query and key vectors during attention computation. The rotation is relative: it encodes not just "this token is at position 47" but "these two tokens are 8 positions apart." Relative encoding turns out to generalize much better to longer sequences than absolute encoding.

Even with RoPE, though, there are limits. If a model is trained with sequences up to 8,192 tokens, the rotation magnitudes it encounters during training span a certain range. Sequences far beyond 8,192 tokens require rotation magnitudes the model has never seen. This causes positional confusion — the model loses track of where things are in the sequence, which is one reason raw "context window" numbers and actual reliable usable context can diverge significantly.

The technique for extending context beyond the training limit is called RoPE scaling. It amounts to rescaling the rotation angles so that a longer sequence maps back into the angular range the model saw during training. Several variants have been proposed — linear scaling, YaRN (Yet Another RoPE extensioN), NTK-aware scaling. These allow deployers to push a model's functional context window beyond what it was trained on, with varying degrees of quality loss at the extended range. Anthropic and OpenAI have not published their exact positional encoding approaches for current frontier models, but the technical reports acknowledge that specific training and fine-tuning procedures enable the long-context ranges their models advertise.

The important conceptual point: a context window number tells you the maximum, not the quality at that maximum. A model may have a 128k-token context window and exhibit meaningfully degraded behavior for content positioned at token 90,000 versus token 500. The ceiling and the quality curve across the window are different things.

The Attention-Cost Tradeoff in Practice

There's another practical limit that the benchmark numbers don't capture: the cost of actually filling the context window.

Anthropic's API pricing for Claude 3 Opus as of early 2025 was $15 per million input tokens. At that price, a single request that uses a full 200k-token context window costs $3.00 in input tokens alone, before a single output token is generated. If your application is making thousands of such requests per day, the arithmetic changes the product economics substantially.

GPT-4 Turbo's 128k-token context at OpenAI's published pricing is in a similar range. Gemini 1.5 Pro's pricing structure is different and cheaper per token, but the compute reality is the same: large contexts are expensive.

This isn't a reason to avoid long contexts. It's a reason to be deliberate about what goes in them. Every token of context you include is paying a price. Tokens that land in the attentional dead zone of the middle — tokens the model demonstrably under-weights — are the worst value for that spend.

A pattern that emerges from this constraint is selective context loading. Rather than stuffing all potentially relevant documents into the context, you run a lighter retrieval or classification step to identify the two or three chunks most likely to be useful, and include only those. You trade completeness for precision. For many tasks — especially narrow question-answering over a known knowledge base — the precision strategy outperforms the completeness strategy on both cost and accuracy.

The phrase "just use the full context" sounds like a shortcut. It often is a shortcut. Just not the good kind.

When Long Context Actually Helps

This chapter has spent several pages on the pathologies of long context. That framing is deliberate — the hype oversells it, so the correction needs to be firm. But long context is not a fiction. There are tasks where it genuinely moves the needle.

Whole-document analysis. If your task is to analyze a 50-page legal contract and identify clauses that conflict with a given policy, you need that entire contract in the context. There's no retrieval trick that reliably catches cross-clause interactions you don't know to look for. Full context wins.

Long-form code generation and editing. Generating or refactoring code across multiple files benefits from the model seeing all the relevant files at once. If function A in file 1 calls function B in file 2, and you're changing the signature of B, the model needs to see both to generate a consistent change. RAG over code chunks fails in exactly the cases where whole-context access would succeed.

Coherent long-form writing. Generating a 15,000-word document with consistent characters, arguments, or terminology benefits from the model attending to its own earlier output. Chunked generation with summary handoffs works but introduces seams. Full-context generation over the entire document so far is cleaner when

it fits.

Conversation memory. For an application where the session is the product — a long-running tutoring session, a negotiation assistant, a customer support flow — keeping the verbatim history in context rather than summarizing it preserves nuance that summaries compress away. This is worth the cost when the nuance matters.

In these cases, long context isn't a blunt instrument; it's the right tool. The key is knowing why you're using it, not reaching for it as the default response to any task involving a lot of text.

What "Reading" a Long Context Doesn't Mean

A phrase that comes up constantly in marketing and in casual developer conversation: the model "read" the document. You paste in a 50,000-word annual report and ask for a summary. The model "reads" the report.

This framing is intuitive but subtly wrong in a way that leads to bad debugging.

A human reads a document sequentially. They allocate attention over time. They re-read. They take notes. They construct an internal model that they can later query.

A transformer processes the entire context simultaneously, in a single parallel operation, and produces a probability distribution over the next token. There is no sequential reading. There is no internal model constructed and stored separately from the weights. There is one forward pass, and whatever representations the layers build during that pass are the entirety of what the model "knows" about your document for the purpose of generating a response.

This has a concrete implication. If you ask the model to summarize a long document and then ask a follow-up question about a specific section, the model does not "go back and re-read" that section. It generates a response based on the forward pass that includes both the original document tokens and your follow-up question tokens. The document is in the context, yes. But the attention the model allocates to various parts of that document is determined entirely by the forward pass dynamics — including the lost-in-the-middle bias we've already discussed.

When you discover that the model "missed" something in a long document, the instinct is often to assume the document was too long and exceeded the context. Sometimes that's true. More often, the information was in the context but positioned poorly, or it was in the center of a long retrieved block and got under-attended. That's a different bug with a different fix.

A Brief Note on Context Caching

One recent architectural addition worth flagging: context caching, which Anthropic and Google have both shipped in different forms.

The idea is that if you're making multiple API calls with the same large prefix — the same 100,000-token system prompt and knowledge base, for instance — you don't need to recompute the transformer activations for that prefix on every call. You compute it once, cache the key-value attention states (the KV cache), and reuse them. Subsequent calls that share the cached prefix only pay for processing the new tokens that differ.

For Anthropic's API as of 2025, cached input tokens were priced at a substantial discount — roughly 10x cheaper than uncached input tokens. For applications with a large, stable system prompt and high request volume, this changes the economics significantly.

Caching doesn't change the attentional dynamics inside the context. The cached prefix still affects the model the same way it would if computed fresh. What it changes is the price you pay to include that prefix repeatedly. It makes the long-context pattern of "large stable knowledge base plus small variable query" much more affordable at scale.

The Honest Mental Model

Here's the version of context windows you should carry around:

A context window is a fixed-size workspace. Everything in the workspace is technically "accessible" during generation. But accessibility is not uniform — the edges of the workspace get more attention than the middle, and the end gets the most, because it's adjacent to where generation happens. The workspace has a hard size limit. Approaching that limit costs real money in compute. And the model has no episodic memory between calls — when the session ends, the workspace is wiped.

Given that mental model, the practical rules write themselves:

Put critical instructions at the start of the system prompt. Put the most important retrieved document last, not buried in the middle. Restate the user's question close to the end of the prompt, right before the model generates. Keep chat history short and recent. Summarize old turns; don't let them accumulate in the center of the context. When you're debugging a "the model ignored X" failure, ask where X was positioned before assuming the context was too short.

And when a new model ships with a 10-million-token context window — this will happen — remember that the number describes a ceiling. It does not describe uniform quality across that ceiling. It does not tell you what the attentional distribution looks like at position 4,000,000 versus position 5. It does not tell you what it costs per call at full occupancy.

The number is real. What you do with it is the engineering problem.

Nelson Liu's 2023 paper is worth reading in full if you deploy retrieval-augmented systems. It's twelve pages, it's freely available on arXiv, and it will change how you order your retrieved chunks. Small structural choices — document order, query repetition, summary placement — produce measurable differences in output quality without changing the model, without changing the retrieval system, without adding any

latency. That's the rare category of optimization: nearly free and well-evidenced.

The context window is powerful infrastructure. Treat it like infrastructure: understand its constraints, design around them, and resist the temptation to throw more tokens at a problem when thoughtful placement would work better.

CHAPTER X

Tool Use and Function Calling

The model does not call your API. Full stop.

Say it once more before continuing, because the naming conventions, the docs, the conference talks, and roughly ninety percent of the blog posts written about this feature all conspire to imply otherwise. OpenAI calls it "function calling." Anthropic calls it "tool use." Neither phrase is technically accurate. The model calls nothing. It writes text. Specifically structured text, formatted to a schema, that a surrounding piece of software interprets as an instruction to call something. Your code fires the HTTP request. Your code handles the response. Your code packs that result back into the next prompt. The model just keeps doing what it has always done: predicting tokens.

That distinction sounds pedantic until the moment your production system fails in a way you cannot explain, or your agent loops forever on a tool that returned an error, or you wonder why giving the model "access" to a database still produces hallucinated rows. The mental model is the architecture. Get it wrong and you'll debug symptoms instead of causes.

This chapter is about the actual mechanism—not the abstraction layered over it.

What the Model Actually Emits

Start with a baseline experiment. Take GPT-4, give it no special tool configuration, and prompt it: "What is the current price of AAPL stock?" It will tell you a number. It will probably be wrong. It may add a hedge—"as of my knowledge cutoff"—or it may not. Either way, the number came from training data, not from a live query.

Now configure the same model with a tool: a function called getstockprice that accepts a ticker symbol as a string. Ask the same question. Instead of a number, the model emits something like this:

``json { "name": "getstockprice", "arguments": { "ticker": "AAPL" } } ``

That is the output. A JSON object. Nothing has been called. No HTTP request has gone out. The model finished its forward pass, the probability distribution over vocabulary was sampled, and the tokens that came out happened to encode valid JSON matching your schema. That is literally it.

Your application receives that JSON. It parses it. It routes getstockprice to whatever implementation you wrote—a yfinance call, a Bloomberg API request, a Redis cache lookup, whatever you've wired up. It gets back a price. It serializes that price into another string—something like {"ticker": "AAPL", "price": 213.42, "timestamp": "2026-05-09T14:32:00Z"}—and appends it to the conversation context as a new message with a special role. OpenAI calls that role tool. Anthropic marks it as a tool_result block. The labels differ; the

concept is identical. The result is now more tokens in the prompt.

The model runs another forward pass over that augmented context and produces a natural-language response incorporating the price. To the user it looks like the assistant "looked up" the stock. What actually happened is a round trip: text in, text out, external execution, text in again.

How the Schema Works

Every tool is defined as a JSON Schema before the model ever sees the conversation. That schema describes the function name, a plain-text description of what it does, and each parameter—its name, type, and description.

The description is not decorative. It is the signal the model uses to decide whether and when to invoke a tool. A schema that says get_weather with a description of "Get the current weather for a location" teaches the model, through fine-tuning and in-context inference, that this tool is appropriate when a user asks about weather conditions. If your description is vague, the model's selection behavior will be vague. The names and descriptions are part of the interface. Treat them like API documentation, because for the model, that is exactly what they are.

Parameter types—string, integer, array, object, boolean—constrain what the model should generate inside arguments. Enums constrain it further. If your unit parameter is defined as enum: ["celsius", "fahrenheit"], the model has a strong prior toward generating one of those two strings. It is not a hard constraint enforced by parsing; it is a probabilistic bias from training on examples where schema constraints were respected. The model can still generate something outside the enum. That is a failure mode to handle in your runtime, not a guaranteed impossibility.

OpenAI's function calling specification, released with GPT-3.5-turbo and GPT-4 in June 2023, established the JSON Schema format that most of the industry has since converged on. Anthropic's tool use API follows the same underlying structure. The Hugging Face Transformers library supports analogous formats for open-weight models that have been fine-tuned for tool use. The vocabulary differs slightly across providers; the principle does not.

The Training That Makes This Work

A base pre-trained model—the raw artifact that emerges from next-token prediction on hundreds of billions of tokens—does not reliably emit well-structured JSON when it would be helpful to do so. It has seen JSON in its training data. It can reproduce it. But it has no learned behavior around when to produce tool-call-formatted output versus when to produce natural language.

That behavior comes from post-training fine-tuning on curated examples. The training corpus for tool-use capability looks something like this: a conversation where a user asks a question, a schema is provided, the assistant response is a correct and well-formatted tool invocation, a tool result is injected, and the assistant's

final response uses that result appropriately. Run enough of those examples—at OpenAI's and Anthropic's scale, "enough" means thousands to tens of thousands of high-quality demonstrations—and the model develops a reliable prior toward that pattern whenever context contains a tool schema and a query that matches one of the defined tools.

RLHF layers over this: human raters (or an automated reward model trained on human preferences) score sampled tool-use outputs on dimensions like whether the right tool was selected, whether arguments were correctly extracted from the user's query, and whether the final answer correctly incorporated the tool result. The model's weights shift toward behaviors that score higher. As Anthropic described in their work on Constitutional AI and in their public model documentation, the post-training process is not a single step but a pipeline—supervised fine-tuning followed by preference learning—and tool use capability is a product of that whole pipeline, not just any one part.

The practical upshot for you, as a developer: the model is not "reasoning" about your schema the way a programmer reads an interface definition. It is pattern-matching to training examples. This matters because it means performance degrades when your schema or your query looks unlike anything in that training distribution. A tool called xfn7 with a description reading "does the thing" will be used less reliably than a tool called searchcustomerrecords with a clear, specific description. The model's behavior is shaped by the statistical regularity of examples it learned from.

Parallel Calls and Multi-Tool Conversations

Early implementations only allowed the model to invoke one tool per response. One call, wait for the result, then continue. Current frontier models—GPT-4o, Claude 3.5 Sonnet and later, and some open-weight fine-tunes—support parallel tool calls: a single response containing multiple tool invocations that can be executed concurrently.

The parallel call feature matters in practice because agentic tasks frequently require several independent lookups. Ask a model to prepare a briefing on a company before a sales call and it might simultaneously invoke searchcompanynews, getcrmrecords, and fetchlinkedinprofile in one pass. Your orchestration layer executes all three in parallel, returns all three results, and the model produces the briefing in one final response. Without parallel calls, this is three sequential round trips; each one adds latency proportional to your tool execution time plus the model's inference time.

The limit on how many tools you can define in a single context—and how large those schemas can be—is a token budget question. Schemas are tokens. A tool definition with a long description and several parameters might cost 200 to 400 tokens. Define thirty tools and you've spent eight to twelve thousand tokens before the user's first word. This matters on models with 8k or 16k context windows. Even on 128k-context models, token costs accumulate in ways that affect both latency and price per call. Prune your tool lists to what is actually relevant to the task.

The Runtime Is Where Everything Happens

The model emits a tool call. Nothing executes until your code runs. This is the part that seems obvious when stated but gets blurred in practice, especially as higher-level frameworks—LangChain, LlamaIndex, Anthropic's Claude agent SDK, OpenAI's Assistants API—automate the dispatch loop.

When a framework "automatically calls" a function, what it is doing is: receiving the model's tool-call output, using a registry you configured, invoking the function in your application process or via an API, and injecting the result back into the context on your behalf. The automation is real and useful. But the model never touches your system directly. No LLM has a socket. No LLM makes a DNS query. The separation is absolute. The model lives inside a GPU kernel; your tools live in your infrastructure.

Why does this matter? Because every error that can happen in that execution layer—network timeouts, authentication failures, malformed API responses, rate limits, exceptions thrown by your function—is your problem to handle in your runtime, not the model's. And because the model does not experience the failure; it reads about it in the next turn. You inject the error message as a tool result. The model then decides, based on that string, whether to retry with different arguments, call a different tool, or apologize to the user. That decision is again a text prediction. It is not error handling in the programming sense.

Design your error result formats as thoughtfully as your success formats. An error string that says "error": "500 Internal Server Error" gives the model almost no signal. An error string that says "error": "No records found for customer_id=9823 — try searching by email instead" gives the model enough context to potentially take a useful next step. You are writing prompts, constantly. Tool results are prompts.

A Concrete Walkthrough

Concrete examples illuminate mechanism better than description. Here is a full loop, numbered step by step.

Turn 1 — User input: "Can you add a meeting with Sarah Chen on Friday at 3pm about the Q2 roadmap?"

Tool schema provided to the model: ``json { "name": "createcalendarevent", "description": "Create a new calendar event. Use this when the user wants to schedule a meeting, appointment, or any time-blocked activity.", "parameters": { "title": { "type": "string" }, "attendees": { "type": "array", "items": { "type": "string" } }, "starttime": { "type": "string", "description": "ISO 8601 datetime" }, "durationminutes": { "type": "integer" } }, "required": ["title", "start_time"] } ``

Model output — Turn 1: ``json { "name": "createcalendarevent", "arguments": { "title": "Q2 Roadmap Discussion", "attendees": ["sarah.chen@company.com"], "starttime": "2026-05-15T15:00:00", "durationminutes": 60 } } ``

The model has extracted the relevant entities from the user's natural-language request—the attendee, the day, the time, the topic—and marshaled them into structured arguments. It guessed a 60-minute default because the user didn't specify duration. It guessed Sarah's email format. Both guesses could be wrong.

Your runtime, step by step:

- Parse the JSON
- Resolve sarah.chen@company.com—either from a contacts lookup or by passing it as-is
- Make the actual API call to Google Calendar (or Exchange, or whatever you use)
- Receive a success response with the event ID and confirmation URL

Tool result injected into context: ``json { "success": true, "eventid": "8kz3p9w", "calendarlink": "https://calendar.google.com/event?id=8kz3p9w", "start": "2026-05-15T15:00:00", "attendees_notified": ["sarah.chen@company.com"] } ``

Model output — Turn 2: "Done — I've added 'Q2 Roadmap Discussion' with Sarah Chen on Friday, May 15th at 3:00 PM. She should receive a calendar invite shortly."

Eleven tokens of structured output. One API call that your code made. One result string. One natural-language confirmation. That is the entire mechanism.

The ReAct Pattern and Multi-Step Chains

Single-tool calls answer single questions. Useful, but limited. Most interesting agent tasks require a sequence of tool calls where the output of one informs the selection and arguments of the next. This is the territory of agentic architectures.

Shunyu Yao and colleagues at Princeton and Google Research formalized one influential approach in their 2022 paper "ReAct: Synergizing Reasoning and Acting in Language Models." The core idea: prompt the model to alternate between a Thought step (natural language reasoning about what to do next) and an Act step (an actual tool call), then observe the result and repeat. The model's reasoning trace becomes part of the context it attends to in subsequent steps, anchoring later decisions in explicit intermediate logic rather than compressed implicit state.

The measurable impact was significant. On the HotpotQA multi-hop reasoning benchmark, ReAct-prompted models outperformed chain-of-thought-only baselines—particularly on tasks requiring information retrieval across multiple sources. The paper showed that the combination of explicit reasoning and tool-grounded action reduced hallucination specifically in cases where the model needed facts it didn't have: instead of confabulating, a ReAct-prompted model would take an action to retrieve, observe the result, and reason from actual data.

Modern frameworks implement variants of this loop automatically. But the underlying structure is always the same: generate, act, observe, repeat. The model never accumulates state outside the context window. Every fact it "remembers" from earlier in the task is there as tokens it can attend to. When the task exceeds the context limit, you must either summarize and compress or accept that earlier observations will fall off. The context window is the working memory. There is no other memory unless you build it.

What Tool Use Does Not Fix

Tool use is often positioned as the solution to hallucination. It is not. It is a reduction mechanism for a specific subset of hallucinations—specifically, the ones caused by the model lacking current or precise factual information. If you give the model a search tool and it retrieves accurate documents, it should (and usually will) produce responses grounded in those documents rather than in stale training data. That is real and valuable.

But it does not fix hallucinations that occur during synthesis—when the model combines retrieved facts in ways that are locally plausible but globally incorrect. It does not fix errors in reasoning about retrieved data. And it does not fix the failure mode where the model calls the wrong tool, constructs incorrect arguments, and confidently incorporates the wrong result. Garbage in, garbage out still applies. The garbage is just coming from a different source.

Retrieval-augmented generation—the broader pattern of injecting retrieved text into context—shifts the epistemic burden from the model's training distribution to your retrieval system and your document corpus. If your search returns the wrong document, or a document that is technically accurate but not relevant to the specific question, the model will often anchor on it anyway. The "lost in the middle" phenomenon documented by Nelson Liu and colleagues at Stanford—where information positioned in the center of a long context receives less attention than information at the edges—applies to tool results injected into long prompts just as it applies to any other positioned content. Put critical retrieved information near the end of the context, close to where the model will generate. Not buried in the middle of a 40,000-token research dump.

Structured Outputs: A Related but Distinct Feature

Tool use and structured outputs are related but not identical. Tool use is specifically about the pattern of: decide whether to invoke an external action, format the invocation, receive the result, continue generation. Structured outputs—forcing the model to emit valid JSON matching a schema for its final response—is a broader constraint that applies whether or not any tool is called.

OpenAI's "JSON mode" and its successor "structured outputs," introduced progressively from 2023 through 2025, use constrained decoding: during sampling, tokens that would violate the schema are assigned zero probability. This is a harder guarantee than "the model was trained to prefer valid JSON." With constrained decoding, the output is provably schema-valid. The tradeoff is that constrained decoding can be slower—the runtime must maintain a parser state alongside the token generation loop—and it requires the schema to be compiled into a format the sampler can use as a constraint mask.

For tool-call arguments specifically, constrained decoding is increasingly available and worth using when you need reliable argument extraction. A model in plain text mode might say "arguments": {"ticker": "AAPL"} with no trailing brace, or use a key name that doesn't exactly match your schema. Constrained decoding prevents both.

Tool Design as Interface Design

Here is something that gets underemphasized in documentation: the quality of your tool definitions directly determines how reliably the model uses them. This is interface design, and the same principles apply.

Be specific about what the tool does and when to use it. If you have two tools that could answer similar queries—say, searchknowledgebase and searchcustomerhistory—write the descriptions to distinguish the cases. "Use this to search the internal knowledge base for product documentation and policy questions" versus "Use this to search the customer's history of purchases, support tickets, and preferences." Ambiguous descriptions produce ambiguous selection behavior.

Keep parameter names self-explanatory. A parameter called q is worse than one called searchquery. A parameter called dt is worse than dateiniso8601format. The model predicts argument values based on how the parameter is named and described, in combination with the user's request.

Provide examples in descriptions when the expected format is non-obvious. If a parameter accepts a date string, note whether you want "2026-05-15" or "May 15, 2026" or a Unix timestamp. The model has seen all three in training. Without a signal, it will pick one. It might pick the wrong one.

And limit the tool list to what is genuinely relevant to the task at hand. An agent configured with forty tools for a task that requires two will still function—but it will sometimes invoke the wrong one, and the schema tokens consume budget that could hold more context. Compose tool sets around task domains, not around "everything we might ever need."

Authentication, Security, and What the Model Shouldn't Touch

The model cannot authenticate to your APIs. It does not hold credentials. It emits text. Your runtime holds the credentials, makes the authenticated call, and returns the result. This separation is not just a technical artifact—it is a security boundary you should design around deliberately.

Do not put API keys, access tokens, or sensitive configuration into the model's context. Not because the model would "misuse" them—the model has no intent—but because whatever goes into the context can appear in generated output. A model that has seen a bearer token in its context window can reproduce it in a later turn if the generation conditions make that plausible. Log carefully. Audit what flows through your tool results. Treat every string that passes through the model's context as potentially appearing in a log, a response, or a downstream prompt.

The principle of least privilege applies to tools the same way it applies to any software service. If the model only ever needs to read from a database, don't wire it to a tool that can write. If it only needs to search a narrow document corpus, don't give it a tool that can traverse your entire file system. The model's selection behavior is probabilistic. Errors happen. Limit the blast radius by limiting what each tool is actually capable of doing.

The Assistants API and Managed Loops

OpenAI's Assistants API, and similar managed execution environments, wrap the generation-tool-execution loop so you don't implement it yourself. You define tools, start a thread, send a message, and poll for the run status. When the run reaches a state requiring tool execution, the API pauses and surfaces the required tool calls; you execute them, submit the outputs, and the run continues. The loop is managed; the execution is still yours.

The abstraction is convenient, especially for production deployments where you want state persistence across sessions without managing the context injection yourself. The tradeoff is opacity: you have less direct visibility into exactly what tokens are in the context at each step, what the model actually emitted before the framework reformatted it, and why the model made a particular tool-selection decision. For debugging and fine-grained prompt optimization, the raw API is often more tractable. For shipping a feature with minimal infrastructure, the managed loop wins.

Anthropic's agent SDK provides a similar pattern, with the execution loop exposed as an iterable stream of events so you can observe each step—model output, tool invocation, tool result—without sacrificing the managed loop structure. The event-stream design makes it easier to build UIs that show real-time agent progress, which matters enormously for user experience when a task spans eight tool calls over thirty seconds.

The Mental Model to Carry Forward

Reduce this whole mechanism to its simplest accurate description and it is this: the model is a text transformer. It reads text and writes text. Tool use trains it to write text in a structured format on specific occasions. Your runtime reads that structured text and does the actual work. The result comes back as more text. The model reads that text and writes a response.

Nothing in that loop requires the model to "understand" what a function does. It does not model causality between the tool call and the result. It reads the result tokens the same way it reads any tokens: as context for predicting the next token in a sequence. It can use tool results extremely effectively—chaining searches, incorporating retrieved facts, adapting arguments based on error messages—because its training included examples of doing exactly those things. But the mechanism is statistical coherence, not comprehension.

This is not a cynical point. The practical capability is genuine. An agent that correctly books a flight, searches a codebase, queries a database, and synthesizes a report from three API responses is doing something useful—dramatically useful. But the correct mental model is that you have a sophisticated text predictor that has been trained to produce tool-call-shaped text at useful moments, paired with orchestration software that turns that text into real actions. The partnership between the two is the thing that works.

Build for that partnership. Write tool descriptions as carefully as you write prompts. Handle tool errors as carefully as you handle API errors. Audit what flows through context as carefully as you audit what flows

through a database. The model is a collaborator with unusual strengths and fixed limitations, and its half of the work is always, only, text.

CHAPTER XI

Reasoning Models and Chain-of-Thought

Here's a question that sounds simple until you try it: what is 17 × 24?

Most people don't know the answer off the top of their head. But write it out — 17 × 20 = 340, 17 × 4 = 68, 340 + 68 = 408 — and you get there. The intermediate steps aren't decoration. They're load-bearing. Your brain can't hold "17 × 24" in working memory and produce "408" in a single mental move, but it can chain three smaller moves together.

Language models have the same structural limitation. A single forward pass through a transformer produces one token. That token is a function of everything in the context window up to that point, but once the forward pass ends, nothing can be added to it. The model can't reconsider. It can't check its arithmetic mid-thought. It either had everything it needed, or it didn't.

Chain-of-thought prompting is the insight that you can fix this by turning reasoning into tokens.

The Paper That Named It

In 2022, Jason Wei and colleagues at Google published "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models" at NeurIPS. The paper's core contribution was not technically complex. They simply showed that if you include worked examples in your prompt — not just (question, answer) pairs, but (question, reasoning steps, answer) triples — large models became dramatically better at tasks requiring multi-step logic.

The benchmark they leaned on hardest was GSM8K: 8,500 grade school math word problems assembled by Cobbe et al. at OpenAI. Not calculus. Not linear algebra. Problems like: A store has 12 apples. Sarah buys 5. The next day, 8 more arrive. How many now? The kind of problem a ten-year-old can solve if they slow down and think.

Without chain-of-thought prompting, PaLM 540B — Google's largest model at the time — solved roughly 17 percent of GSM8K problems with standard few-shot prompting. With chain-of-thought examples in the prompt, the same model, same weights, no retraining, hit roughly 58 percent. That's not a marginal improvement. That's the difference between a coin flip with a head start and something actually worth deploying.

GPT-3 175B showed similar gains, jumping from around 17 percent to 46 percent on the same benchmark. The trick worked across arithmetic, commonsense reasoning, and symbolic manipulation. And the same weights produced the improvement — no fine-tuning, no architectural change. Just a different prompt format.

Why This Works, Mechanically

Here is where it helps to remember what a transformer actually does at inference time.

Each token generated becomes part of the context. The next token attends to all previous tokens when computing its representation. That's the architecture. There is no separate memory buffer, no stack, no register. There is only the context window and what's in it.

When you ask a model to answer directly — "What is the sum of the first 15 odd numbers?" — you're asking it to compress an entire multi-step derivation into a single output token or a very short sequence. The intermediate computation has to happen implicitly, inside the model's forward pass, distributed across billions of floating-point multiplications. Some of that works. A lot of it doesn't, especially when the correct answer requires coordinating many sub-results that don't naturally co-occur in training data.

When you ask the model to show its work first, something different happens. "The first odd number is 1. The second is 3. Their sum is 4. The third is 5, so the running total is 9..." Each of those statements becomes a token sequence in the context. When the model later produces the final answer, it isn't doing implicit arithmetic — it's attending to an explicit intermediate result that it already wrote out. The answer is downstream of attended context, not upstream of implicit computation.

The clearest metaphor: scratch paper. The scratch paper doesn't give you new capabilities. It offloads working memory onto a physical medium you can refer back to. Tokens are the model's scratch paper.

You Don't Even Need Examples

Wei et al. used few-shot prompting — worked examples in the prompt demonstrating the reasoning format. But also in 2022, Takeshi Kojima and colleagues at the University of Tokyo showed something even starker: you can get chain-of-thought behavior with zero examples, using just the phrase "Let's think step by step."

Append that sentence to a math problem and GPT-3 produces a reasoning trace before the final answer. Don't append it and it doesn't. The phrase appears to function as a cue that activates a generation pattern the model has seen abundantly in training data — textbooks, worked solutions, tutoring forums — where problems are followed by step-by-step explanations. The model has seen this pattern millions of times. The prompt is pointing at it.

Zero-shot CoT underperforms few-shot CoT on most benchmarks. But the fact that it works at all is informative. It suggests the model isn't learning to reason from your prompt examples at inference time — it already has latent capability for structured intermediate generation, and the prompt is unlocking it.

For developers: if you're building a system and can control the prompt, you don't need to hand-craft elaborate CoT examples for every task. A well-placed "think through this carefully before answering" captures most of the benefit. Task-specific few-shot examples with worked steps get you further, but the gap is smaller than you might expect.

The Scale Tax

One of the most important findings in Wei et al. is the one that doesn't make headlines: chain-of-thought prompting only works reliably above a certain model size.

Small models — anything under roughly 100 billion parameters in the 2022 experiments — produced "reasoning chains" that were often incoherent, circular, or simply wrong. The final answers weren't better. Sometimes they were worse, because the flawed reasoning chain polluted the context the answer was attending to. You got the scratch paper, but the scratch paper was covered in errors.

Above 100 billion parameters, the gains became consistent and large. Below that threshold, the model doesn't have reliable enough internal representations of arithmetic or logical operations to produce coherent intermediate steps. Its "reasoning" looks superficially structured but is mostly confabulation — right format, wrong substance.

The practical implication is direct: if you're running a smaller model and adding chain-of-thought prompting doesn't help, it might genuinely not be able to use the technique rather than you applying it wrong.

Reasoning Models Are Different

Everything above covers inference-time prompting: take a pre-trained model and change the format of your input to elicit better behavior. Chain-of-thought prompting is a prompting technique. Reasoning models are something else.

OpenAI's o1, released in September 2024, was the first widely deployed model explicitly trained to reason. Not trained with CoT examples in the prompt. Trained — via reinforcement learning — to produce extended internal reasoning traces as part of its generation process. The distinction matters.

When you send a message to o1, the model generates a long sequence of "thinking" tokens before producing its visible response. These thinking tokens constitute a structured internal monologue: the model considers approaches, checks intermediate results, notices errors, revises, and produces a final answer once it's satisfied. OpenAI shows users a summarized version of this process rather than the raw tokens, but the generation is real — real compute, real tokens, real cost.

The key word is trained. OpenAI used reinforcement learning to reward o1 for producing correct final answers on hard reasoning problems. The model learned that producing longer, more careful intermediate reasoning improved its chance of reward. The behavior emerged from the training process, not from a prompt engineer appending "let's think step by step."

This means o1's internal reasoning is qualitatively different from zero-shot CoT applied to a model that wasn't trained for it. The model has internalized what useful intermediate steps look like for hard problems. It isn't just mimicking the format of reasoning — it has been selected, through training, for producing reasoning that actually leads to correct answers.

The AIME 2024 results are the clearest quantitative demonstration. The American Invitational Mathematics Examination targets the top five percent of high school math students in the United States. GPT-4o, on a single attempt, solved roughly one or two of the twelve problems. O1, on a single attempt, solved roughly nine or ten. That is not a marginal difference. It's the difference between a model that can't handle advanced competition math and one that mostly gets through it.

Extended Thinking: Anthropic's Approach

Anthropic's equivalent is called extended thinking, first made available in Claude 3.7 Sonnet in early 2025. The mechanism is similar: the model generates a block of thinking tokens before producing its response, and those tokens are longer and more computationally expensive than standard generation.

The developer-facing difference from OpenAI's approach is transparency. Anthropic exposes the thinking content to developers rather than summarizing it. When you enable extended thinking via the API, you get back a thinking block alongside the response block. You can read the model's intermediate reasoning directly. This is genuinely useful for debugging — you can see whether the model correctly identified the structure of a problem, where it went wrong, and whether its stated reasoning matches what it actually did. That last check is informative in ways that might surprise you.

Both approaches share the concept of a thinking budget. You specify a maximum number of tokens the model can spend on internal reasoning before producing a response. Higher budget: potentially better answers on hard problems, definitely higher latency and cost. Lower budget: faster and cheaper, but the model has less scratch paper.

Anthropic's technical reports describe extended thinking as particularly beneficial for multi-step math proofs, complex coding problems, and tasks requiring explicit constraint satisfaction — cases where a standard generation pass is likely to drop a constraint or miss a sub-step. The gains are less consistent on tasks that don't require multi-step chaining. On those, additional thinking time buys almost nothing.

What "Thinking Tokens" Actually Are

Let's be precise, because the marketing language obscures the mechanics.

When o1 or Claude with extended thinking generates "reasoning," the same forward-pass process is running as any other generation. The model generates a token. That token is added to the context. The model generates the next token, attending to everything including the just-generated token. Repeat, for potentially hundreds or thousands of steps.

The difference is that this generation happens before the visible response begins. The thinking tokens form an internal context that the final answer then attends to. Architecturally, it's identical to chain-of-thought prompting — tokens generated early in the sequence inform tokens generated later — but the model is trained to use this space for genuine intermediate computation rather than relying on the user to elicit it through

prompt engineering.

There is no separate reasoning engine. No logic module. No symbolic planner attached to a neural network. The transformer is generating text. Some of that text looks like intermediate reasoning steps, and that helps it get the final tokens right.

This is worth stating plainly because the product marketing around "reasoning models" frequently implies something more exotic. O1 is not reasoning in any philosophical sense. It's not constructing proofs from axioms or running a search algorithm over logical states. It's generating tokens, and some of those tokens happen to be intermediate reasoning steps, and that reliably improves the final output on hard problems. That's real and useful. It's also not magic.

The Budget Control Problem

One genuinely novel engineering question that reasoning models surface: how much thinking is enough?

For a simple factual question, additional thinking tokens are mostly wasted. The model knows the answer — or doesn't — and will arrive at it quickly. Forcing it to generate 2,000 tokens of internal reasoning before answering "What year was the Eiffel Tower completed?" doesn't improve accuracy. It just burns time and money.

For a hard math proof or a complex debugging task, the model may genuinely need a long reasoning chain to get the answer right. Cutting the budget too short means it runs out of scratch paper midway through the derivation and has to guess the rest.

The problem is that you usually don't know in advance which situation you're in. Anthropic's technical documentation acknowledges this directly: extended thinking works best when the task genuinely requires multi-step chaining, and the right budget depends on task complexity in a way that's hard to specify ahead of time. OpenAI's o-series handles this internally, with the model allocating its own compute budget based on the apparent difficulty of the question.

The practical engineering approach is to benchmark your specific task distribution. Run a sample of your inputs through extended thinking at several budget levels. Measure accuracy against ground truth. Find the point where additional budget stops meaningfully moving accuracy. That's your operating point. Don't guess — the curve varies substantially by domain.

What Chain-of-Thought Doesn't Fix

This is the part the enthusiastic blog posts tend to skip.

Chain-of-thought prompting and reasoning models improve accuracy on tasks where the bottleneck is multi-step computation — carrying intermediate results forward, checking sub-results, coordinating multiple constraints. They do not fix the underlying statistical nature of the model's knowledge.

If the model has a false belief encoded in its weights — a fact that appears more frequently or plausibly in training data than the truth — CoT doesn't correct it. The model will reason fluently from a false premise. The reasoning chain will look coherent. The conclusion will be wrong. This is particularly dangerous because a confident-looking chain of reasoning can feel more trustworthy than a direct wrong answer. The model is wrong and shows its incorrect work, and the incorrect work is convincing.

It's routine. Ask a reasoning model a question that touches on a domain where training data had significant noise — niche scientific claims, events at the edge of its training cutoff, specific numerical values that vary by source — and you'll see confident, multi-step reasoning arriving at an incorrect answer. Every step looks valid. The conclusion doesn't hold up.

There's also a compounding problem. A model that is 95 percent reliable at each individual reasoning step is only about 77 percent reliable across a five-step chain, assuming independent errors. The errors in real reasoning chains are not independent — they're correlated, because early mistakes shape the direction of later ones. The more steps, the worse the math.

And there's the question of what counts as a "reasoning task" in the first place. Models perform better on chain-of-thought style problems when those problems resemble the format of problems in their training data. Math competition problems have a well-established genre in human-generated text. The model has seen millions of worked solutions. The reasoning chain it produces is partly genuine computation and partly pattern matching against that genre. Move to a genuinely novel problem structure — one that doesn't map onto a recognizable template — and the benefit of extended thinking decreases.

The Reliability Gap in Practice

A software engineer at a mid-sized startup told me they evaluated o1 against GPT-4o for a pipeline that required parsing legal contract terms and then applying a multi-step logic tree to classify clauses. GPT-4o with standard prompting got about 61 percent accuracy on their test set. GPT-4o with few-shot CoT got to 74 percent. O1 reached 89 percent. Real, useful progress — the difference between a tool that mostly doesn't work and one that earns a place in a production pipeline.

But 89 percent is not 100 percent. At their throughput, the remaining 11 percent of incorrectly classified clauses still required human review. The reasoning model reduced the review burden by more than half. It didn't eliminate it. They deployed it. It was worth it. And they didn't pretend the problem was solved.

That's the calibrated view. Reasoning models are models that have learned to use generation-time computation more efficiently by producing explicit intermediate context. On hard multi-step problems, they outperform CoT prompting on a base model. CoT prompting on a base model outperforms direct prompting on that same base model. Each layer is a real improvement.

None of them constitute reliable step-by-step reasoning in any formal sense. None of them guarantee accuracy. None of them eliminate hallucination. They shift the distribution toward better outputs without collapsing the variance to zero.

For Developers: When to Use What

The decision tree is not complicated.

If you're using a standard instruction-tuned model and your task involves multi-step logic, calculation, or constraint satisfaction: add chain-of-thought instructions. "Think through this step by step before giving your final answer" is a fine starting point. Few-shot examples that demonstrate reasoning in your specific domain get you further. Don't skip this for complex tasks — the performance delta is large enough that it's almost always worth the additional output tokens.

If you're using a reasoning model like o1 or Claude with extended thinking: your prompting style should change. These models are already trained to reason — prompting them to "think step by step" can actually interfere with their trained behavior. Let the model use its reasoning budget on the problem. Give it a clear problem statement and constraints. Don't try to control the structure of its intermediate thinking. That's already trained in.

Budget the thinking tokens to match task complexity. Don't set a maximum thinking budget of 8,000 tokens for a task where your hardest problems resolve in 1,000. Don't cap at 500 when you're asking the model to audit a complex multi-file codebase. Measure where the accuracy-versus-cost curve flattens and operate there.

Verify the outputs on your specific task. AIME scores and MATH dataset accuracy reflect performance on well-defined problems with clear ground truth. Your production task is probably neither as clean nor as well-matched to the training distribution as those benchmarks. The only way to know how much reasoning models help you is to measure it on your data, with your success criteria, against your baselines.

The mechanism, stripped of the mythology: intermediate tokens are scratch paper. Training on reasoning tasks teaches models to use that scratch paper productively. The marketing language around "thinking" and "reasoning" describes real behavior — just not the exotic kind.

It's still the same transformer, doing the same thing it always does, generating one token at a time.

The question is how well it learned to use those tokens before the answer.

CHAPTER XII

Multimodal: Vision, Audio, and the Rest

Text is easy. Text is already discrete—characters become tokens, tokens become vectors, vectors flow through the transformer in a clean pipeline you've read about in every prior chapter. The whole architecture was designed around text. Vaswani et al.'s 2017 paper doesn't mention images once.

But the world is not discrete text. Contracts are scanned PDFs. Dashboards are screenshots. Customer complaints arrive as voice recordings. Medical records include X-rays. If an LLM can only process text, it can only help with a fraction of the actual information that exists—and developers building real products have been bumping into that wall since the first GPT APIs shipped.

Multimodal models are the answer. Not a different kind of AI—the same transformer architecture, extended to handle non-text inputs by converting them into something the transformer already knows how to process: sequences of vectors. The elegance of the solution is worth understanding precisely, because it also explains the limitations.

The Core Insight: It's All Just Vectors

Start here, because everything else follows from it.

A text token is a vector. A 768-dimensional or 4096-dimensional or whatever-dimension float array. The transformer doesn't care that it originally represented the word "contract." It sees numbers. The attention mechanism computes relationships between those numbers. The feed-forward sublayers transform them. The whole machine operates on vectors.

So the question for multimodality is not "how do we teach the transformer to understand images?" The question is narrower and more tractable: "how do we convert an image into a sequence of vectors that live in the same numerical space as text token vectors—so the transformer can apply attention across all of them together?"

That's it. That's the whole trick. Once you have that, the transformer's existing machinery works unchanged. No new architecture. No special image-reading subroutine. Just vectors, wherever they came from.

Vision Encoders: Slicing Images into Patches

The dominant approach to vision comes from a 2020 paper by Dosovitskiy et al. at Google Brain: "An Image is Worth 16x16 Words." The title tells you the method. Take an image. Cut it into a grid of small rectangular patches—16×16 pixels is the classic choice, though implementation details vary. Flatten each patch into a vector. Project that vector through a learned linear transformation into the same dimensionality as your text

token embeddings. You now have a sequence of patch embeddings that look, mathematically, identical to token embeddings.

A 224×224 image at 16×16 patches produces 196 patch vectors. A Vision Transformer—ViT, in the literature—feeds those 196 vectors into a standard transformer encoder and processes them with attention, exactly as text tokens are processed. The model learns during training what each patch means relative to every other patch and relative to any accompanying text. No hand-crafted features. No convolutional filters manually designed to detect edges or gradients. Just patches, projections, and attention.

Modern multimodal LLMs extend this. GPT-4V and GPT-4o use a vision encoder that processes images at potentially much higher resolutions, breaking them into tiles first. OpenAI's technical documentation describes a tiling strategy where a high-resolution image is split into 512×512 pixel tiles, each tile independently encoded into roughly 170 tokens, with an additional low-resolution thumbnail providing global context for another 85 tokens. A 1024×1024 image at high detail therefore consumes approximately 765 tokens before you've typed a single word of your prompt. A 2048×2048 image—say, a detailed engineering diagram—can consume over 2,000 tokens on its own.

That number matters. We'll come back to it.

Gemini took a different approach from the start. Rather than bolting a separately trained vision encoder onto a text-trained language model, Google trained Gemini natively multimodal from the beginning—text, images, audio, and video data processed jointly during pre-training. The Gemini technical report describes this as a "multimodal mixture of encoders" approach where image patches are processed by purpose-built encoders whose outputs are interleaved with text tokens from the first training step. The practical consequence: Gemini doesn't have the slight impedance mismatch that arises when you fuse a separately trained vision encoder with a separately trained language backbone. Whether that architectural difference produces better downstream performance on specific tasks is an empirical question that varies by task, and you should measure it rather than assume.

What the Model Is Actually Doing With an Image

Here is where precision of language matters.

The model does not "see" the image. Seeing implies perceptual experience—light hitting retinas, signals traveling optic nerves, visual cortex activating. None of that happens. What happens is this: the image's pixel values are transformed through a series of matrix multiplications into a sequence of vectors. Those vectors enter the attention mechanism alongside text token vectors. Attention weights determine how much each image-patch vector influences each text token vector, and vice versa.

When you ask GPT-4o "what is shown in this photo?" and it produces an accurate description, it isn't describing what it sees. It's producing the statistically most plausible continuation of a sequence that contains image-patch vectors and the text tokens of your question. The patch vectors encode spatial and color information that was useful during training for predicting text descriptions of similar patches. The model

generates the description because descriptions of images are what followed image-like vector patterns in the training data.

That's not a lesser thing. It's genuinely powerful. But it's also why the model fails in specific, predictable ways—and understanding the mechanism tells you where those failures will be.

Audio: Spectrograms as Images

Audio is harder than vision in one specific way: it's temporal. A 10-second audio clip is 10 seconds of continuously varying pressure values sampled at, say, 16,000 Hz—that's 160,000 individual samples. You can't just flatten that into a vector; it's too long and too raw.

The standard solution converts audio into a spectrogram—a two-dimensional representation where one axis is time, the other is frequency, and the value at each cell represents the energy of that frequency at that moment. Specifically, most modern audio processing uses mel spectrograms, which space frequency bins according to the mel scale, roughly matching human auditory perception of pitch. OpenAI's Whisper, which underpins audio in GPT-4o, uses 80-channel mel spectrograms computed over 30-second audio chunks.

Here's the useful insight: a mel spectrogram is an image. It's a rectangular grid of values, just like an image is a rectangular grid of pixel values. So the same patch-and-project approach that works for vision works for audio. You cut the spectrogram into patches. You project them into the embedding space. You get a sequence of audio-patch vectors that the transformer can attend to alongside text.

The audio encoder in Whisper is a convolutional network followed by a transformer encoder—a hybrid that first extracts local features from the spectrogram via convolution, then processes those features with attention. The output is a sequence of embeddings that represent the acoustic content of the audio chunk. Those embeddings are then fed into the language model decoder as additional context, alongside any text.

GPT-4o made the audio pipeline native rather than a pre-processing step. Earlier approaches piped audio through Whisper to get a text transcript, then fed the transcript into the language model. That works but loses prosody, tone, and any information that doesn't survive transcription—pauses, emphasis, overlapping speech. GPT-4o processes audio tokens directly, which is why it can comment on someone's tone of voice or notice that a speaker is hesitant. Whether those observations are reliable is a separate question.

Injection Into the Transformer Sequence

Let's make this concrete with a simple example.

You send GPT-4o a message: an image of a bar chart, followed by the text "Which quarter had the highest revenue?"

At the model's input layer, your message is processed into a unified sequence of vectors:

- The image is encoded into, say, 340 patch vectors (low-detail mode, 85 tokens × 4 for a standard encoding).
- Your text is tokenized into something like 9 tokens: "Which", "quarter", "had", "the", "highest", "rev", "enue", "?" (tokenization splits "revenue" in practice).
- A special token or embedding marker indicates where the image ends and text begins—different implementations handle this differently, but the model needs to learn which parts of the sequence correspond to which modality.

Now the transformer has a sequence of roughly 349 vectors. Attention operates over all of them. Every text token can attend to every image patch. Every image patch can attend to every text token. The multi-head attention mechanism—as Vaswani et al. describe for text—applies identically: each position queries all other positions, the compatibility scores are softmaxed into weights, and the weighted sum of value vectors becomes the position's updated representation.

There is no special image-processing pass. No separate vision pipeline running in parallel. One unified forward pass, one attention computation, one sequence. The integration is architectural, not procedural.

This is also why multimodal context windows get expensive fast. Text tokens are cheap because tokenizers are efficient—English prose averages roughly 0.75 words per token. Image patch tokens are expensive because images contain a lot of spatial information that doesn't compress the same way. The context window limit is measured in total tokens regardless of modality. A model with a 128,000-token context window sounds generous until you fill 2,000 of those tokens with a single image.

Two Tokens, One Sequence: Why Alignment Is the Hard Part

The vision encoder and the language backbone are trained separately, or semi-separately. They start with different objectives—the vision encoder learns to represent image patches in ways useful for image tasks, the language model learns to predict text. To make them work together, their output spaces need to be aligned.

The common solution is a learned projection layer—sometimes called a connector or an adapter—that maps the vision encoder's output vectors into the language model's embedding space. This projection is trained on paired image-text data: image captions, alt text, screenshots with associated descriptions, interleaved text and image content from the web.

CLIP, OpenAI's Contrastive Language-Image Pre-Training model, established a key technique here. CLIP trains two encoders—one for text, one for images—with a contrastive objective: push the embeddings of matched image-text pairs close together, push mismatched pairs apart. The result is a shared embedding space where an image of a golden retriever and the text "a golden retriever" end up near each other in vector space. This aligned space is useful as a foundation for the projection layer that connects vision encoders to language models.

The alignment is learned, not perfect. It generalizes across the distribution of image-text pairs seen during training. Outside that distribution—unusual visual styles, domain-specific diagrams, charts with non-standard

formatting—the alignment weakens, and the model's outputs about the image become less reliable.

Where Multimodal Inputs Actually Help

Some tasks are genuinely better with multimodal input. Not marginally better—categorically better, where the text-only alternative either doesn't exist or is far more cumbersome.

Document understanding. A PDF is not just text. It's layout, tables, headers, figures, annotations, signatures. Parsing a scanned contract through OCR and feeding the extracted text to an LLM loses the spatial structure that often carries meaning. Which paragraph is a header versus body text? Which numbers are in a table and which are in a footnote? A multimodal model processing the PDF as an image can use spatial position as evidence. For structured document extraction—pulling specific fields from invoices, insurance forms, or lab reports—vision-capable models outperform text-only pipelines on realistic messy documents. By a lot, in practice.

Chart and graph interpretation. Asking a user to describe a chart in text before asking a question about it is impractical. Asking a multimodal model to interpret a bar chart, line graph, or scatterplot works reasonably well for standard chart types with clear labels. The model doesn't reconstruct the data numerically—it produces text that describes the chart the way a caption writer would. For downstream tasks like "does this chart show an upward trend?" or "which category is largest?", that's often sufficient.

UI and screenshot analysis. Testing automation, accessibility evaluation, and user experience research increasingly involve sending screenshots to LLMs for structured feedback. "Does this button have sufficient contrast?" or "What is the error message shown?" are questions that only make sense with an image input.

Code with visual output. A developer writing matplotlib code can paste the generated chart back into the conversation and ask "why is the legend overlapping the data?" rather than describing the problem in text. The tight feedback loop between code generation and visual inspection is genuinely useful.

Multimodal audio applications. GPT-4o's native audio processing enables applications that text-based pipelines can't match: real-time voice conversations with sub-500ms latency, translation that preserves the speaker's original tone, analysis of customer service calls for sentiment and escalation signals.

Where It Still Fails

The honest list is long. Longer than marketing materials suggest.

Counting. Ask a model to count the number of people in a crowd photo. Or the number of red dots in a diagram. Or the number of shelves in a bookcase. Performance degrades badly past single-digit counts. The patch-based encoding doesn't preserve precise spatial counting information—patches blend local content into vectors that are good for recognition and description but poor for exact enumeration. A 2023 study examining GPT-4V on basic counting tasks found error rates climbing sharply above five or six objects.

Fine-grained spatial reasoning. "What object is directly to the left of the red box?" Sounds simple. Reliably wrong often enough to be a production risk. The attention mechanism learns global relationships well but struggles with the kind of precise relative spatial reasoning humans find trivial. The model may describe the image accurately at a high level while getting specific positional relationships wrong.

Small text in images. Low-resolution screenshots, photos of printed documents, whiteboard captures—text at small point sizes frequently comes back garbled or hallucinated. The patch-based encoding loses fine detail at small scales. If precise text extraction from an image is your use case, a dedicated OCR system will outperform a general multimodal LLM consistently.

Complex domain-specific diagrams. Electrical schematics, organic chemistry structural formulas, UML diagrams, circuit board layouts. These require specialized visual vocabularies that are underrepresented in general web training data. The model may confuse similar-looking symbols, miss the significance of topology, or produce plausible-sounding but incorrect interpretations.

Temporal reasoning in video. Video is the frontier that's still mostly unsolved. You can sample frames and send them as images, but that loses temporal dynamics—motion, speed, causality across time. Some models accept video tokens directly, but the context costs are severe (a 60-second video at 1 frame per second, each frame encoded at 85 tokens minimum, is already 5,100 tokens with no text yet), and the reasoning about what changed between frames and why remains weak relative to how easily humans process the same video.

Audio with multiple speakers. Overlapping speech, cross-talk, speaker diarization—assigning who said what in a multi-person conversation—is still an active research problem. Native audio models handle single-speaker audio well. Difficult acoustic environments, non-native accents at the margins of the training distribution, and complex multi-speaker recordings all degrade performance significantly.

Token Cost Is Not a Footnote

I mentioned it earlier and I want to make it sharp, because developers routinely underestimate this.

Every image patch vector that enters the context window occupies space that costs money and takes time. An OpenAI API call with a 1024×1024 image in high-detail mode consumes approximately 765 input tokens just for the image. At pricing in effect through 2025, GPT-4o charged roughly $2.50 per million input tokens. A pipeline processing 10,000 documents per day with one high-detail image each generates 7.65 billion image tokens per day—at that pricing, roughly $19,000 per day in input token costs alone, before any text in the prompt.

That math is not hypothetical. It's the kind of math that turns a proof-of-concept into a budget crisis. The practical response involves several levers. Low-detail mode—a single 85-token representation of the image—costs far less and is often sufficient for tasks that don't require fine-grained inspection. Image resizing before submission dramatically reduces tile count. Task-specific routing—using a cheaper vision model for simpler classification tasks, reserving the expensive frontier model for complex analysis—is standard production architecture.

Latency follows the same logic. More tokens to process means a slower forward pass. A simple text-only prompt might return a first token in under 300 milliseconds. The same prompt with a large high-resolution image attached can take two to four seconds before the first token appears. For conversational applications where latency is user-visible, that gap matters.

Building With Multimodal Models: The Practical Frame

If you're integrating vision or audio into a production system, the questions that actually determine success are not about whether the model "understands" images. They're operational:

What specific claims will the model make based on the image, and how do you verify them? If your application extracts a total from an invoice image and that total flows into a payment system, you need a validation step—regex against a known format, cross-check against OCR output, or human review on low-confidence cases. The model will confabulate invoice totals sometimes. Not often. Often enough.

What happens when the image is bad quality? Real-world images from user uploads are blurry, skewed, poorly lit, and partially occluded. Your system needs to handle degraded image quality gracefully—detect it and ask for a better image rather than silently producing unreliable output.

Do you actually need image tokens, or do you need OCR? For extracting text from standard forms, dedicated OCR pipelines with layout analysis (Tesseract, Amazon Textract, Google Document AI) are faster, cheaper, and more accurate than passing the image directly to a multimodal LLM. Use the right tool. A multimodal LLM adds value when the task requires reasoning about the image, not just reading it.

Where should the image appear in the context? The "lost in the middle" degradation documented in research on long text contexts applies to multimodal contexts too. Images and highly relevant text placed at the beginning or end of a long prompt receive more reliable attention. Images buried deep in a long multi-turn conversation may receive less weight than their relevance warrants.

Are you certain about the training distribution for your image type? A model trained heavily on consumer photography and web screenshots will process those modalities reliably. The same model processing satellite imagery, electron microscope outputs, or specialized medical scans is operating further from its training distribution. Expect weaker performance and design your confidence thresholds accordingly.

The Honest Current State

Multimodal capability has moved from a novelty to a production reality in roughly three years. GPT-4V's release in late 2023, GPT-4o's native multimodal architecture in 2024, Gemini's integrated multimodal training from initial release—these shifted what a production LLM can do. The architectural story is clean and elegant. The performance story is messier.

For the tasks where it works—document layout understanding, chart description, screenshot analysis, voice interaction—multimodal adds real capability that would be difficult or impossible to replicate with text-only

systems. For the tasks where it fails—precise counting, small-text extraction, complex spatial reasoning, video temporal analysis—the failures are structural. They follow from how patch-based encoding works and how attention aggregates information. They won't be solved by making the model larger. They require either different architectural choices or task-specific mitigations.

The model doesn't see. It doesn't hear. It operates on vectors derived from pixels and waveforms, the same way it operates on vectors derived from text. That frame is more useful than the anthropomorphized version—not because it's philosophically purer, but because it tells you exactly where to expect the system to succeed and where to build guards around its failures. A developer who understands that the model processes 170-token tile vectors from image patches will architect their invoice extraction pipeline differently than one who assumes the model "reads" the invoice the way a human does.

Understanding the mechanism is how you use the tool well.

CHAPTER XIII

Cost, Latency, and Why 'Just Use the Biggest Model' Is Wrong

There is a default setting in most developers' heads when they start building with LLMs. It goes something like this: pick the biggest, most capable model, ship it, and only revisit the choice if someone complains about the bill. It's understandable. Frontier models are genuinely impressive, and when you're in the early prototyping phase, paying a few dollars in API credits to prove a concept costs less than the time required to think carefully about model selection. Fine. Prototypes are prototypes.

Production is different.

At production scale, the "just use GPT-4" instinct stops being pragmatic and starts being expensive in ways that compound: financially, in latency, and in the hidden architectural debt you accumulate when you never build the evaluation infrastructure to know whether the big model is actually necessary. This chapter is about breaking that instinct — not by telling you to always use cheap models, but by giving you the precise, concrete framework to know when the frontier model earns its price and when it's quietly burning your budget for no measurable gain.

The Four Real Drivers of Inference Cost

Before any pricing discussion makes sense, you need to understand what you're actually paying for. LLM inference cost is a function of four things, and conflating them leads to bad decisions.

Input tokens. Every token you send to a model — your system prompt, the user's message, any retrieved documents, any conversation history — costs money on input. This is often underweighted by developers who focus on output length because output feels like "the work," but input can dominate at scale. A 2,000-token system prompt sent 100,000 times per day is 200 million input tokens monthly. At $15 per million tokens for a frontier model's input pricing, that's $3,000 a month just from a prompt you wrote once and forgot about.

Output tokens. Generation is typically more expensive than input, per token, because it is computationally more intensive. During input processing, the model runs tokens through the transformer in parallel. During generation, it produces tokens one at a time — each new token requires another forward pass through the entire network. On OpenAI's pricing as publicly documented through mid-2025, GPT-4o charged roughly $2.50 per million input tokens and $10 per million output tokens — a 4:1 ratio. On Anthropic's Claude 3 Opus, the gap was steeper: $15 per million input, $75 per million output. Output tokens are the expensive side. If your use case involves generating long documents, that asymmetry matters enormously.

Model parameter size. This is the physics underneath the pricing. A model with 70 billion parameters requires dramatically more GPU memory and compute per forward pass than a model with 7 billion. Serving a 70B model requires multiple high-end GPUs just to hold the weights in VRAM, before processing a single token. Smaller models fit on fewer GPUs, parallelize more cheaply, and generate tokens faster. The pricing tiers offered by API providers are not arbitrary marketing categories — they reflect real differences in hardware requirements. When Anthropic prices Claude 3 Haiku at roughly $0.25 per million input tokens versus $15 for Opus, that 60x price difference maps to a genuine 60x (or more) difference in the compute resources deployed to serve that request.

Infrastructure and batching efficiency. This matters more if you're running your own models, but it shapes API pricing too. A single user making occasional requests has terrible hardware utilization — the GPUs sit mostly idle between calls. A provider serving thousands of concurrent requests can batch them efficiently, keeping utilization high and amortizing the fixed hardware costs across more tokens per hour. This is why self-hosting a frontier model for a low-volume internal tool almost always costs more than using the API: you're paying for peak capacity, not average load.

The Math You Should Do

Pick a representative workload. Concrete numbers make this real in a way abstractions never do.

Say you're building a customer-support triage system. Each request comes in with a fixed system prompt (800 tokens) plus the user's message (average 120 tokens). The model needs to classify the issue and suggest a response template — call it 200 output tokens per request. You expect 50,000 requests per day.

Daily token counts:

- Input: (800 + 120) × 50,000 = 46,000,000 tokens
- Output: 200 × 50,000 = 10,000,000 tokens

Now run the numbers across model tiers, using approximate public API prices:

| Model | Input price ($/M) | Output price ($/M) | Daily cost | Monthly cost | |---|---|---|---|---| | GPT-4o (OpenAI, mid-2025) | $2.50 | $10.00 | $215 | $6,450 | | GPT-4o mini | $0.15 | $0.60 | $13 | $387 | | Claude 3 Opus | $15.00 | $75.00 | $1,440 | $43,200 | | Claude 3 Haiku | $0.25 | $1.25 | $24 | $720 |

The monthly cost swings from $387 to $43,200 for the identical task — a 112x difference. And that's before accounting for the fact that the cheap models might handle the triage classification just as well as the expensive ones, because classification is a well-scoped, relatively simple task where the incremental capability of a frontier model buys you almost nothing.

This is the core argument. The expensive model earns its price on tasks where its additional capability translates into better outcomes on your specific task. For many tasks in production — classification, entity extraction, summarization with clear criteria, code completion in a narrow domain — the cheaper model is not a downgrade. It's the correct engineering choice.

Latency Is a Separate Axis

Cost and latency are related but not the same thing, and optimizing for one does not automatically optimize for the other.

Latency has two components that matter differently depending on your use case. Time to first token (TTFT) is how long the user waits before seeing anything — critical for interactive applications, irrelevant for batch pipelines. Total generation time is the full time from request to complete response — critical when you're chaining model calls and each one adds to the wall-clock time of a pipeline.

Smaller models are almost always faster on both axes. They have fewer transformer layers, fewer attention heads, fewer parameters to compute through per forward pass. A 7B or 13B parameter model on capable hardware can produce tokens at 80–120 tokens per second. A 70B model on the same hardware might produce 15–25 tokens per second. Multiply that across a multi-step agent pipeline with five sequential model calls, and the latency difference becomes minutes, not seconds.

There's a subtler latency trap in context length. All else equal, processing a 100,000-token context takes longer than processing a 5,000-token context — attention is computed over every token pair in the sequence, and that computation scales with sequence length. Developers who reach for maximum context size as a substitute for careful prompt design often pay for it twice: in tokens billed and in latency added. Stuffing an entire knowledge base into the context because it's available is rarely the right move. It's a shortcut that defers the work of figuring out what the model actually needs.

When Smaller Models Win

The intuition that bigger is better comes from benchmark comparisons. On MMLU, HumanEval, and GPQA-style evaluations, frontier models consistently outperform smaller ones. That's real. The mistake is assuming benchmark rankings transfer cleanly to any specific production task.

They don't. Here's why.

Benchmarks measure breadth and ceiling. They test whether a model can handle a wide range of diverse, often difficult questions drawn from many domains. Your production task, in almost every case, is not a benchmark. It is narrow. It is repetitive in its structure. It has a specific format requirement. It operates in a domain where 95% of requests look like some variation of the same thing.

A fine-tuned 7B model trained specifically on customer support tickets from your industry will outperform GPT-4o on your customer support tickets. Not because it's a "smarter" model in any general sense, but because it has been optimized specifically for the distribution of your task, while GPT-4o has been optimized for an extremely broad distribution that includes physics proofs and Swahili poetry and bug fixes in Rust.

Meta's Llama 3 8B, available as an open-weight model, achieves performance competitive with GPT-3.5-level models on many standard benchmarks — at zero marginal API cost if you self-host it. Mistral's

7B base model, released in late 2023, outperformed Llama 2 13B on every benchmark Mistral reported, despite having roughly half the parameters. The relationship between parameter count and capability is not linear, and it's certainly not a reliable proxy for performance on your specific task.

The empirical question you need to answer is not "which model scores highest on leaderboards?" It's "which model scores highest on a representative sample of my actual inputs, judged by my actual success criteria, at what cost and latency?" That's a different question entirely, and it requires building an evaluation — something Chapter 14 covers in detail. The point here is that skipping the evaluation and defaulting to the largest model is not engineering. It's just assuming.

Smaller Doesn't Mean Worse on Scoped Tasks

Let me be specific about what "well-scoped" means, because it's doing a lot of work in this argument.

A scoped task has:

- A narrow input distribution (customer messages from one industry, legal clauses of one type, code in one language)
- A clear output format (a JSON object, a category label, a fixed-template response)
- An evaluable success criterion (output matches a known reference answer, passes a unit test, gets rated above threshold by human raters)

Sentiment classification is scoped. Named-entity extraction from medical records is scoped. Summarizing earnings call transcripts into a fixed JSON schema is scoped. Answering "write me a poem about my startup in the style of Pablo Neruda" is not scoped — it's open-ended creative generation where the frontier model's broader training and stylistic range genuinely matters.

In a 2023 evaluation published by researchers at Stanford studying LLM performance on specialized tasks, smaller fine-tuned models consistently matched or exceeded GPT-4 on domain-specific classification tasks while requiring a fraction of the compute. The pattern holds: when the task is narrow enough to be represented well in fine-tuning data, a smaller model trained on that distribution beats a large model operating on priors from its massive general training.

Even without fine-tuning, smaller models often hit adequate performance on structured tasks with good prompting. "Adequate" matters. If your success criterion is 95% accuracy on a classification task and GPT-4o delivers 97% while GPT-4o mini delivers 96%, the frontier model is not providing meaningful value for 15x the price. You're paying a premium for 1 percentage point of headroom you may not need.

Model Routing: Production Architecture, Not a Hack

If your application receives a mix of request types — some simple, some complex — routing those requests to different models based on their difficulty is not a workaround. It is a legitimate, mature architectural pattern.

The intuition is straightforward: not every request deserves the same model. A user asking "what are your business hours?" and a user asking "explain the legal implications of this contract clause in relation to the EU AI Act" have radically different complexity profiles. Sending both to your most expensive model wastes money on the first request. Sending both to your cheapest model may fail on the second.

Model routing builds a layer between the application and the models that classifies incoming requests and dispatches them to the appropriate tier. In its simplest form, this is a small classifier — possibly another language model, possibly a rule-based system, possibly an embedding-similarity lookup against a labeled request library — that assigns each incoming request to a tier. Simple requests go to the fast, cheap model. Complex requests escalate to the capable, expensive one.

RouteLLM, a routing framework released publicly in 2024 by researchers at UC Berkeley and collaborating institutions, demonstrated that routing between a strong and a weak model could reduce costs by 85% while maintaining output quality within 5% of always using the strong model on a range of benchmark tasks. The key insight is that most queries in practice are not at the frontier of difficulty. If your user base is typical, a substantial fraction of requests are repetitive, low-complexity, or structurally similar to things the cheaper model handles reliably. Routing lets you capture that efficiency without sacrificing quality on the tail of hard requests.

There are a few common routing strategies:

Classifier-based routing. A trained classifier takes the incoming prompt and predicts which model tier should handle it. The classifier can be as simple as a logistic regression on prompt embeddings or as sophisticated as a small LLM fine-tuned to evaluate request complexity. The cost of the routing classification needs to be factored into your economics — running a cheap embedding model for every request is negligible; running GPT-4o to decide whether to run GPT-4o is not.

Confidence-based routing (cascading). Send every request to the cheap model first. If the cheap model's output confidence exceeds a threshold, return it. If not, escalate to the expensive model. This works well when the cheap model can express calibrated uncertainty — which it sometimes can, through logprob signals on constrained output formats, and sometimes cannot, for open-ended generation. Cascading adds latency on escalated requests because you're running two sequential calls, so it's best suited for batch or near-real-time use cases where latency is secondary to cost.

Rule-based routing. Some routing decisions don't need a model at all. If a request comes in with a document attachment larger than 50,000 tokens, it goes to the context-efficient model. If a request contains the string "legal" or "compliance" in the system tag, it escalates automatically. If it's a simple FAQ lookup because a retrieval system already returned a high-confidence match, it goes to the cheapest generation tier or returns the retrieved passage directly. Hard rules are brittle but cheap to compute and easy to audit.

In practice, production systems combine all three. The rule layer filters obvious cases. The classifier handles ambiguous ones. Cascading catches the edge cases the classifier misjudged. You build the sophistication of the routing layer in proportion to the cost savings it unlocks — at low volume, a simple rule is enough; at

high volume, a more precise classifier pays for itself quickly.

Model Tiering as Architecture

Routing implies tiers, and tiers imply that you've thought carefully about what tasks belong in each tier. This requires more discipline than it sounds.

A common tiering structure:

Tier 1 (fast, cheap, high-volume). Small models in the 7B–13B parameter range or equivalent API-priced small model endpoints. Use for: classification, entity extraction, short-form structured output, simple retrieval augmentation over a small context, FAQ responses. Target latency: under 500ms to first token. Cost target: sub-$1 per million input tokens.

Tier 2 (mid-range, versatile). Models in the 30B–70B range or mid-tier API endpoints. Use for: longer-form summarization, moderate-complexity reasoning, multi-turn conversation with context tracking, code generation in well-supported languages, document-level analysis. Target latency: 1–3 seconds to first token. Cost target: $1–5 per million input tokens.

Tier 3 (frontier, capable, expensive). The largest available models. Reserve for: tasks that genuinely require frontier capability — complex multi-step reasoning, novel code synthesis in niche domains, nuanced judgment on subjective criteria, advanced multimodal tasks. Target latency: acceptable to be 5+ seconds. Cost: $10–75 per million output tokens depending on model.

The critical discipline is not assigning tasks to Tier 3 by default and only moving them down when they complain. Do the reverse. Start with Tier 1, evaluate, escalate only when performance is genuinely inadequate, and require evidence — not intuition — for every escalation decision. Most teams that do this discover that less than 20% of their production traffic actually needs Tier 3.

The Infrastructure Overhead You Didn't Budget For

One more cost category that often surprises teams that move from API-only to self-hosted or hybrid architectures: infrastructure overhead is not proportional to your token volume in a linear way.

A self-hosted 70B model requires, at minimum, four to eight A100 80GB GPUs to hold the weights in VRAM at float16 precision. As of 2025, renting that hardware on any major cloud provider costs between $12 and $20 per GPU per hour. At eight GPUs, you're spending $100+ per hour whether you receive one request or ten thousand. If you're running this to save money over API pricing, you need sustained throughput of several million tokens per day before the economics favor self-hosting at all. Below that threshold, the API is almost always cheaper on a total-cost basis once you account for engineering time, redundancy, and ops.

Quantization helps. A 70B model quantized to 4-bit precision can fit on two or three A100s with modest quality degradation — often acceptable for specific tasks. Techniques like GPTQ and GGUF quantization, popularized through the llama.cpp ecosystem, make this accessible. But "accessible" doesn't mean "free of

complexity." You still need the ops infrastructure, the monitoring, the serving layer, and someone whose job it is to keep the thing running.

The honest accounting: for teams under roughly one billion tokens per month of sustained usage, commercial APIs are almost certainly cheaper than self-hosting when you price in engineering time honestly. Above that scale, or when you have data-privacy requirements that preclude sending data to third-party APIs, self-hosting becomes worth evaluating. Not before.

Model Selection Is an Engineering Decision

Here is the reframe this chapter has been building toward: model selection is not a prestige decision. It is an engineering trade-off, identical in kind to choosing a database or a message queue. You don't pick PostgreSQL because it's the most powerful database that exists. You pick the database that fits your access patterns, your consistency requirements, your team's operational familiarity, and your budget. The choice is right or wrong relative to the task, not in the abstract.

LLM model selection works the same way. The correct selection process looks like this:

- Define your task precisely. What goes in, what comes out, what makes an output good.
- Build a representative evaluation set. Fifty to two hundred representative inputs, labeled with ground truth or evaluated by your success criterion.
- Run the evaluation on two or three candidate models across price tiers.
- Pick the cheapest model that meets your performance threshold.
- Build monitoring to catch performance drift as model versions change.

Step 4 is the one most teams skip, and it's where the money leaks. "Meets your performance threshold" requires knowing what your performance threshold is — which requires having done step 2. Most teams never do step 2, which is why most teams end up overpaying.

There's a second dimension here beyond cost: vendor lock-in and model version stability. A model you depend on today may be deprecated or quietly changed six months from now. OpenAI has updated GPT-4 Turbo multiple times with changes to behavior that weren't always announced with much fanfare. Anthropic has introduced new Claude versions that shift performance in subtle ways. If your evaluation infrastructure is solid, you catch these regressions. If it isn't, you find out the hard way when users complain.

Open-weight models — Llama, Mistral, Qwen, DeepSeek — have a different risk profile. The weights don't change unless you update them. You control the deployment. But you absorb the operational cost and lose access to the continuous improvement that API providers bake in. There's no universally correct answer. There is a correct answer for your specific situation, derived from the specifics of your task, your traffic volume, your data sensitivity requirements, and your team's capabilities.

The Prestige Trap

A brief word about psychology, because it's real.

There is social pressure in engineering teams to use the frontier model. Saying "we use GPT-4o" or "we're on Claude 3 Opus" carries a kind of credibility signal in demos and investor conversations that "we're on Claude 3 Haiku with routing to GPT-4o mini" does not, even though the latter is almost certainly the more sophisticated engineering decision. The prestige gradient runs upward toward the expensive model, and that gradient pulls decisions in ways that don't show up in the architecture docs.

The antidote is making cost visible. Dashboard the token spend per feature, per route, per model tier. When the engineering team can see that one particular endpoint is spending $4,000 a month and a 45-minute evaluation sprint revealed the cheap model performs identically on that endpoint's actual inputs, the decision to switch becomes obvious rather than contentious. The conversation changes from "are we confident enough in the cheap model?" to "what's the evidence that we need the expensive one?" That's the right question. Evidence goes in the direction of capability claims, not away from them.

Cost visibility also reveals optimization opportunities that prestige-first selection misses entirely. If you can see that your average input token count per request is 2,400 tokens and 1,600 of those are a boilerplate legal disclaimer appended to every prompt regardless of context, trimming that disclaimer saves 67% of your input cost immediately — no model change required. Token efficiency on the input side is almost always the fastest win, and it only becomes visible when you're measuring.

The Number That Matters

You are not building the most impressive LLM-powered product. You are building the most effective one — effective for your users, your use case, and your economics. Effectiveness is measured by whether it does what it's supposed to do, not by which model name appears in the API call.

The benchmark number that actually matters for your production system is not a model's MMLU score. It's your task-specific accuracy, latency, and cost per successful output. Three numbers. Measured on your data, with your success criterion, running against the models you're evaluating, in the serving configuration you'll actually deploy.

Everything else is marketing.

Start cheap. Measure rigorously. Escalate only when the data says you must. Route intelligently when your traffic has meaningful complexity variance. Build the evaluation infrastructure before you need it, not after.

That's the field manual version of model selection. It's less exciting than "use the most powerful model available," and it will save you enough money over a year of production traffic to fund whatever exciting thing comes next.

CHAPTER XIV

Evaluation: How Do You Actually Know a Model Is Better?

Every few months, a new model ships and the benchmarks look incredible. Ninety-something percent on MMLU. State-of-the-art on HumanEval. Top of the Chatbot Arena leaderboard. The blog post is confident, the charts trend up and to the right, and you're left with one question that the announcement never quite answers: will this model actually be better for what I'm building?

That question turns out to be much harder than it looks. Not because the researchers are lying — they generally aren't — but because "better" is always relative to a task, a population of inputs, and a success criterion, and public benchmarks are designed to measure something generalizable, not something specific to your product. The gap between "best on benchmarks" and "best for you" is where a lot of expensive engineering mistakes live.

This chapter maps the evaluation landscape honestly. What the major benchmarks actually test, how they break, what LLM-as-judge gets right and where it quietly lies to you, why human preference ratings are valuable and treacherous at the same time, and how to build a production eval that tells you something true.

The Benchmark Ecosystem: What It Measures and What It Doesn't

Start with MMLU — the Massive Multitask Language Understanding benchmark, introduced by Dan Hendrycks and colleagues at UC Berkeley in 2020. It consists of 57 subject areas, roughly 15,000 multiple-choice questions spanning high school biology through professional law and medicine. A model answers each question by picking A, B, C, or D. Score is fraction correct.

The appeal is obvious. MMLU is cheap to run, easy to compare across models, and covers a wide surface area. When GPT-4 hit 86.4% on MMLU at launch in 2023, that number meant something: the model was doing well on a huge variety of factual recall and basic reasoning tasks that humans take years to learn.

But look carefully at what that number hides.

Multiple choice is not the same as open-ended generation. A model can score 80% on MMLU while being completely unable to write a coherent explanation of any of the concepts it just "got right." The format eliminates the hardest part of most real tasks — producing a structured, accurate, coherent response from scratch. You're testing recognition, not generation. These are related skills, not identical ones.

The second problem is contamination. MMLU questions, along with the answers, have been on the internet since 2020. Models trained on data scraped through 2023 or 2024 have almost certainly seen large fractions of that question set during pre-training. When a model scores 90% on MMLU, you genuinely cannot tell how

much of that is generalization and how much is memorization of the test. Researchers at the Allen Institute for AI documented this contamination problem in detail in 2023, showing that overlap between benchmark datasets and training corpora is widespread and systematically underreported. The benchmark publishers are not at fault; the training data scale makes contamination nearly unavoidable.

HumanEval, released by OpenAI in 2021, takes a different approach. It's a set of 164 Python programming problems where the model must generate a function body given a docstring, and correctness is judged by running unit tests against the output. Pass@k measures whether at least one correct solution appears in k samples. This is meaningfully better than multiple choice — you're checking actual output against ground truth.

The problems, though, are short. Introductory-to-intermediate difficulty. Self-contained. They test whether a model can write a function to count vowels or reverse a linked list, which is useful, but not whether it can implement a feature inside an existing codebase with dependencies, idiomatic conventions, and ambiguous requirements. Most real coding tasks are the second kind.

GPQA — Graduate-Level Google-Proof Q&A — is a more recent attempt to push past contamination. The idea: hire actual PhD-level domain experts to write questions so hard that Google searching for the answer doesn't reliably help, and that even other domain experts struggle. Released by researchers at NYU and scale AI in 2023, GPQA's "diamond" subset of 198 questions sits at roughly 65% accuracy for human experts who aren't specialists in the exact subfield. When Claude 3 Opus launched in early 2024, Anthropic reported it scored around 50% on GPQA diamond — impressive relative to prior models, but still below non-specialist PhD performance on genuinely hard questions.

What GPQA buys you is harder to game. The questions are obscure enough that memorizing them at training time is impractical at scale. What it doesn't buy you is any evidence about whether a model is good at your task. PhD-level chemistry trivia and production code generation have essentially nothing in common.

This is the fundamental issue with every static benchmark: generalizability runs in both directions. A benchmark designed to be general enough to apply to many models is, by definition, not specific enough to apply to your problem. The question "which model scores highest on a fixed academic test set" answers a different question than "which model performs best on customer support tickets in English and Brazilian Portuguese for a B2B SaaS platform."

Leaderboards and the Illusion of Ranking

The Chatbot Arena leaderboard, maintained by the LMSYS team at UC Berkeley, takes a completely different approach. Two model outputs are shown to a human rater side by side, the rater picks the one they prefer, and an Elo-style ranking accumulates from millions of such comparisons. This sidesteps contamination entirely — you're collecting fresh human preferences across a huge variety of organic prompts. GPT-4 and Claude models have consistently ranked near the top; Llama 3 and Qwen 2.5 have surprised people by ranking closer to frontier proprietary models than their parameter counts might suggest.

Chatbot Arena is genuinely useful for getting a rough sense of which models people prefer in freeform conversation. It's not useful for knowing which model is best at structured data extraction, or at following a specific output format reliably, or at staying within a defined persona. The arena's strength — natural, uncurated prompts — is also its weakness. It's measuring aggregate preference on a self-selected population of prompts from people who visit the LMSYS website, which skews toward technical users writing varied open-ended questions in English. That population may or may not resemble your users.

There's also the question of what raters optimize for when they pick a preferred response. Research on human preference annotation consistently finds that raters favor longer, more confident-sounding, better-formatted responses — independent of factual correctness. A model that writes in a more authoritative tone and uses bullet points will tend to beat a technically accurate but plain response in head-to-head preference. This is well-documented in the InstructGPT paper from OpenAI (2022) and in subsequent work: the reward signal from human raters is noisy, systematically biased, and influenced by surface features that have little to do with task success.

So you have static benchmarks that measure the wrong thing at scale, and human preference ratings that measure the right thing but through a distorting lens. Neither one tells you what you actually need to know. What does?

LLM-as-Judge: Useful, Dangerous, and Widely Misunderstood

The past two years produced a fast-growing answer to evaluation cost: use a capable model — usually GPT-4, Claude Opus, or Gemini Ultra — to judge the outputs of the model you're evaluating. Show the judge a prompt, the model's response, and a rubric, and ask it to score. This scales cheaply, can run nightly, and produces structured scores that track over time.

When LLM-as-judge works, it works because a capable judge model has good priors about what a coherent, relevant, well-reasoned response looks like. For tasks where human judgment is straightforward — did this response correctly summarize the document? Does this code output match the specification? — a strong judge model tends to agree with humans at rates in the 70–85% range, depending on the task and how carefully the rubric is written. That's useful.

The failure modes are specific and predictable. Judge models show position bias: when two responses are presented for comparison, judges favor whichever response comes first, at rates well above chance. They show verbosity bias, the same one human raters show. They show self-enhancement bias — GPT-4 acting as judge tends to rate GPT-4 outputs more favorably than human raters do, and Claude acting as judge does the same. If you use the same model family to judge as you're evaluating, you are not getting an independent assessment. You are getting a favorable reference.

Judge models also fail on tasks that require ground-truth knowledge the judge doesn't have. If a model hallucinates a legal citation and the judge model was trained on the same internet corpus, the judge may not catch the error — it has no special access to truth. It's applying plausibility heuristics, which is exactly what

the model being judged is also doing. Using one plausibility engine to check another one's plausibility gives you nothing but the illusion of validation.

Use LLM-as-judge with tight rubrics, a judge model from a different family than the one you're evaluating, and always validate your judge setup against a human-labeled gold set before trusting its scores. Treat the scores as a signal, not a verdict.

Human Annotation: The Ground Truth Problem

Human preference annotation is the closest thing to ground truth the field has. The InstructGPT work used it to train the reward model in RLHF. Anthropic's Constitutional AI work extended it. Scale AI, Surge, and a constellation of annotation platforms have built businesses around it. Done carefully, it tells you what real people actually prefer or find useful, which is what you ultimately care about.

Done carelessly, it introduces systematic error that propagates all the way into your conclusions.

Rater agreement on quality is surprisingly low for subjective tasks. Two independent raters looking at the same pair of model outputs will agree on which is better somewhere around 60–70% of the time on open-ended tasks — better than chance, but far from the 90%+ we'd want from a reliable measuring instrument. Agreement improves dramatically when tasks are specific: "did the model correctly identify all entities in this sentence" gets 90%+ agreement; "which response is more helpful" gets 62%. If your eval task is the first kind, human annotation is solid. If it's the second kind, you need to be humble about what the numbers mean.

Annotation is also slow and expensive. A single round of labeling 500 prompt-response pairs with qualified annotators — people who actually understand the task domain — can cost thousands of dollars and take two weeks. That timeline makes it impractical for catching regressions between model versions when you're shipping every two weeks. Which is why human annotation and automated evals need to coexist: annotation builds and validates your ground truth; automation watches for drift against that ground truth continuously.

One practical note: don't underestimate task specification. The single biggest source of variance in human annotation isn't rater disagreement — it's underspecified instructions. Raters who aren't given a precise rubric will substitute their own judgment, which varies. A rubric that says "rate helpfulness from 1 to 5" produces noisier data than a rubric that says "a score of 5 means the response directly answers the user's question, uses no more than three sentences, and mentions no irrelevant products." Write the rubric like you're writing a unit test.

Building Your Own Production Eval

This is the part that actually matters for your work. Everything above is context for understanding why you can't outsource evaluation to a leaderboard. Here's how to do it yourself.

Start with your inputs, not a benchmark.

Your first step is collecting a representative sample of real prompts your system handles. Not prompts you wish it handled. Not prompts that seem hard. The actual distribution: what are users typing, in what proportions, in what languages, with what kinds of errors and ambiguities? If you're in production, pull 500 real examples from logs. Redact any PII. If you're pre-launch, generate representative inputs by writing 50 yourself and then using a model to generate variations — but validate that the variations match your expected distribution before using them.

This matters more than almost any other decision. An eval dataset that doesn't represent your input distribution is not measuring your system. It's measuring some other system on paper.

Define success before you run anything.

"Better" is not a criterion. Before you run a single model against your dataset, write down what a correct response looks like for each type of input. This means per-category success criteria, because a summarization task and a classification task don't have the same ground truth structure.

For classification tasks: exact match against a gold label. Clean and unambiguous.

For generation tasks: you need to choose. Either write reference answers and score similarity (with the understanding that there are many correct ways to say the same thing), or define behavioral criteria and score against those — the response must contain X, must not contain Y, must be under Z words. The second approach is almost always more useful in practice, because "similarity to reference" can be gamed by verbose paraphrasing while completely missing the point.

For code tasks: execute the code. Run your test suite. Anything short of execution-based validation is trusting the model's confident presentation more than its actual output.

Write these criteria before you see any model output. If you write them after, you will unconsciously write criteria that favor the model that generated the outputs you reviewed first. This is not a personal failing; it's how human cognition works under uncertainty. Lock the rubric first.

Beware of training data contamination in your eval set.

If your eval set contains text that appeared on the internet before your target model's training cutoff, you may be measuring memorization. For off-the-shelf models like Claude or GPT-4, you cannot audit their training data to check. You can partially protect against this by using examples that are internal to your organization — actual customer inputs, proprietary documents, your product's specific jargon — since that content is unlikely to be in any model's training set. Another approach is to use recent data: inputs from the last three months, referencing recent events or version numbers, are less likely to be contaminated than inputs from 2021.

For tasks where you're generating synthetic evaluation data, vary the exact wording significantly from any public examples you used as inspiration. A model that has seen "What is the capital of Australia?" ten thousand times will answer Canberra without actually having to reason; the same model may behave

differently if the question is phrased in a way it hasn't encountered.

Size the eval set to the signal you need.

A 50-example eval cannot tell you whether model A is 3% better than model B. It can tell you whether model A falls apart completely on a type of input that model B handles fine. Sample size and the magnitude of difference you care about are mathematically linked: if your baseline accuracy is 85% and you want to detect a 2-point improvement to 87%, you need roughly 1,500 examples to hit 80% statistical power at standard significance thresholds. If you just want to know whether the new model broke something obvious, 200 examples is often enough.

Be honest about this. A lot of production evals are undersized not because anyone calculated that they don't need more power, but because 500 labeled examples felt like a lot. Run a power calculation. If you don't have enough data to detect the effect size you care about, your eval will produce ambiguous results that you'll then interpret through whatever prior you walked in with.

Track regression, not just absolute score.

A model upgrade that improves average accuracy by 4 points but degrades performance on a specific input category by 15 points is not a straightforward win. Aggregate metrics hide this. Slice your evaluation by category, input length, language, and any other dimension that matters to your use case, and track each slice separately across versions.

This sounds obvious but is routinely skipped. What usually happens instead: the aggregate number goes up, the upgrade ships, and three days later someone notices that a specific customer segment is getting much worse responses. Then you're doing a rollback under production pressure instead of a planned analysis.

Set up your evaluation pipeline to produce slice-level reports automatically. If you're tracking ten slices and aggregate score, and any single slice degrades by more than some threshold you've defined in advance, flag it — don't average it away.

Running the eval: practical mechanics.

For automated scoring, you need three things: the prompt, the model's response, and a score. The score comes from one of your defined criteria — an exact match, a classifier you trained, or an LLM-as-judge with a tight rubric. Run each model candidate against the full dataset before looking at results. Use fixed random seeds for any sampling parameters. Log everything: the model version, the sampling parameters, the timestamp, the raw response text, and the score.

For tasks where you're using LLM-as-judge, run each example through the judge twice with the candidate response in different positions to catch position bias. If the scores disagree, flag the example for human review instead of averaging. Use a different model family for the judge than for the candidate being evaluated.

For human review, use a structured annotation interface — not a spreadsheet, not a Google Form with five fields. A proper annotation tool forces the rubric in front of the rater for every example, prevents skipping

criteria, and logs timestamps. Timing data is useful: raters who complete each annotation in under ten seconds are almost certainly not reading carefully.

Avoid benchmark drift.

Your eval dataset becomes less useful over time for a subtle reason: as you make decisions based on eval results, you implicitly optimize toward doing well on that eval. This is Goodhart's law operating at the system design level. After six months, your prompts and system design may have been tuned specifically toward the weaknesses your eval was designed to catch, making the eval a less accurate measure of genuine capability.

Refresh your eval dataset regularly. Every quarter or two, sample 100 new examples from recent production traffic, label them, and add them to the set — while retiring an equal number of older examples. This keeps the eval distribution fresh and reduces the risk of overfitting your system to a fixed test set.

For open-source models you're fine-tuning, the risk is more direct: if any examples from your eval set appear in your fine-tuning data, your eval results are meaningless. Keep a hard partition between training and evaluation data and enforce it with a deduplication check at the data pipeline level, not as an afterthought.

What "State of the Art" Actually Means in Practice

Here is an uncomfortable truth worth sitting with: state of the art on public benchmarks is essentially a marketing metric. That's not a cynical statement. It's just that benchmark performance is what research teams can publish and what sales teams can point to, so it gets optimized. The optimization is real — models do generally improve on benchmarks over time — but the optimized quantity is not the same as performance on your task.

The practical implication is that you should never make a production model selection decision based primarily on benchmark rankings. Use the rankings to generate a shortlist of candidates worth testing. Then run those candidates against your eval set, with your prompts, under your latency and cost constraints, and pick the one that wins on your numbers.

This is more work. It's also the only way to actually know.

A concrete example of why this matters: a mid-2024 comparison between GPT-4o and Claude 3.5 Sonnet on coding benchmarks showed fairly similar numbers, with slight variations depending on the benchmark. But for tasks involving long document processing with strict output format requirements — a common production pattern — the models behaved very differently in practice, and which one "won" depended heavily on the specific format requirements and document types in play. No public benchmark told you that. Only running the comparison on representative data told you that.

The Eval Is the Product

There's one more thing worth saying, and it's not technical.

Evaluation is often treated as a support activity — something you do to validate the work, not the work itself. This is backwards. For any serious LLM deployment, the evaluation framework is as important as the prompts, the retrieval pipeline, or the model selection. It's the instrument that tells you whether everything else is working. A team with a weak eval running a great model is flying blind. A team with a rigorous eval running a mediocre model can iterate to a good outcome, because they can measure what they're iterating toward.

Invest in evaluation the same way you invest in testing infrastructure in conventional software development. It pays for itself in fewer production incidents, cleaner model upgrade decisions, and — most importantly — actual evidence when someone asks whether the system improved.

The field will keep producing better benchmarks. MMLU will be replaced by something harder, and that something harder will eventually be contaminated too. Leaderboards will get more sophisticated. LLM-as-judge pipelines will improve. None of that changes the core principle: the only evaluation that tells you whether a model is better for your application is an evaluation designed specifically for your application, run on inputs that actually resemble what your users send, scored against criteria you defined in advance.

Everything else is a useful prior. Not a substitute.

CHAPTER XV

The Current Model Landscape in 2026

Here is a caveat you should read before anything else in this chapter.

Every specific number that follows—context lengths, parameter counts, benchmark scores, pricing—reflects publicly available information as of early 2026. By the time you read this, at least one major model family will have released a new version. At least one leaderboard will have reshuffled. At least one company will have announced something that makes a paragraph here look quaint. That is not a failure of this chapter. It is the condition of this field. The goal here isn't to give you a set of facts to memorize. It's to give you a framework for thinking about a landscape that will not sit still.

With that on the table, let's look at what's actually out there.

The Three-Tier Structure

The model landscape in 2026 is not flat. It organizes itself, pretty cleanly, into three distinct tiers defined by how the models are distributed, not just how capable they are.

Tier one is proprietary API models. You don't run these. You call them. The provider hosts everything—the weights, the serving infrastructure, the GPUs—and you pay per token. Anthropic, OpenAI, and Google dominate this space. These models have the most consistent access to frontier compute, which generally shows in their raw capability ceilings, but you are permanently renting, never owning.

Tier two is open-weight models. The weights are publicly downloadable. You can run them yourself, fine-tune them, quantize them, inspect them. "Open-weight" is not the same as "open source"—the training data, the training code, and sometimes the license terms are not fully open—but the weights being available changes everything about deployment flexibility and cost structure. Meta, Mistral, Alibaba's Qwen team, and DeepSeek are the key players.

Tier three is specialized fine-tunes. These are models—usually built on a tier-two base—that have been further trained for a narrow domain: code generation, legal document review, medical triage, customer support in a specific industry. They rarely top a general benchmark. On a well-scoped task, they frequently beat frontier models at one-tenth the cost.

Understanding which tier a model belongs to matters more than any individual benchmark score. It determines your deployment options before you write a single line of code.

Tier One: Proprietary API Models

Anthropic Claude

Anthropic's model family, as of early 2026, runs on the Claude 3 and Claude 3.7 generation. The important distinctions are not just size but capability profile.

Claude 3.5 Sonnet became a practical workhorse for developers throughout 2024 and into 2025—strong on code generation and instruction following, fast enough for interactive use, and priced in a range that didn't require a finance conversation every time you wanted to build something. The 200,000-token context window is real and, more importantly, actually usable. Unlike some models that technically accept long contexts but degrade badly on recall past a certain point (more on that in chapter nine), Claude's long-context performance held up reasonably well on document retrieval tasks.

Claude 3.7 Sonnet, released in February 2026, added what Anthropic calls extended thinking—a budget-controlled internal scratchpad that the model uses before producing its final answer. The mechanism is similar in spirit to what chapter eleven describes for reasoning models generally: more computation at inference time, expressed as visible intermediate steps. The practical effect on multi-step reasoning and complex coding tasks is measurable. The practical effect on latency is also measurable, and not in the direction you want when you're building a chat interface. Extended thinking is the right tool for some tasks and actively wrong for others. Choosing between thinking enabled and thinking disabled is a real engineering decision, not a default.

Anthropic's public positioning leans heavily on alignment and safety research, which is real work—Constitutional AI is a documented methodology, not a marketing slogan—but it's worth being clear-eyed: these safety properties are trained behaviors, not guarantees. The model still hallucinates. It still has off-distribution failure modes. The training approach changes the profile of failures, not their existence.

Licensing: API access only. You don't get the weights. Anthropic's usage policies prohibit a specific list of applications (weapons development, certain surveillance uses, a few others), and those terms can change.

OpenAI

OpenAI's lineup in 2026 splits into two distinct product lines with different architectural philosophies.

The GPT-4o family is the direct-generation line. GPT-4o ships with a 128,000-token context window, handles text, image, and audio inputs in a single unified architecture, and remains one of the most widely deployed frontier models by raw API call volume. GPT-4.1, announced in early 2025, pushed the context window toward one million tokens—making it one of the longest-context proprietary models available—and improved instruction-following measurably on long-document tasks. The pricing structure for million-token contexts is not trivial, but the capability itself is real.

The o-series is different. o1, o3, and the more recent o4 variants are reasoning-first models. They generate longer internal chains of thought before producing output, controlled by a compute budget the user partially influences through the reasoning_effort API parameter. Chapter eleven covers the mechanics in detail, but the

short version is this: on tasks like competition math, complex code synthesis, and multi-step scientific reasoning, the o-series outperforms the GPT-4o line by a margin that is hard to argue with. On tasks like summarizing a meeting transcript or writing a product description, the extra inference time buys you almost nothing. Deploying the wrong model to the wrong task is an expensive mistake in both senses—slower, and more expensive per token.

OpenAI's multimodal story is further along than most. GPT-4o handles interleaved text and images in the same forward pass, produces audio natively rather than through a bolted-on text-to-speech layer, and has a functional real-time API for low-latency voice applications. These capabilities come at a cost premium. Whether that premium is justified depends entirely on your use case.

Licensing: API access only. The model weights are not public. OpenAI's terms of service have evolved significantly since 2022; always read the current version before building a production application.

Google Gemini

Google's model family benefits from one structural advantage that Anthropic and OpenAI simply don't have: infrastructure. Google designed Gemini to run on TPU pods at a scale that changes the economics of inference, and that shows in what they can offer at the Flash tier.

Gemini 2.5 Pro, as of early 2026, sits near the top of most coding and reasoning benchmarks. The 1-million-token context window is the longest natively available from any production API at this tier. In practice, the "lost in the middle" degradation described in chapter nine applies here too—Gemini handles long contexts better than most, but "better than most" is not the same as "uniformly excellent across a million tokens."

Gemini 2.0 Flash is the model worth paying close attention to if you're building something with high throughput requirements. It's fast. It's cheap. It handles multimodal inputs. On many practical tasks, its outputs are difficult to distinguish from much larger models. The Flash line is Google's clearest answer to the question "what does good-enough look like at scale?"

Native multimodality is where Google has invested most heavily. Gemini processes video frames, audio, images, and text in a unified context. Not in the sense that it converts them all to text first—the architecture handles cross-modal attention at the token level. For applications involving document images, slides, or video-grounded Q&A, this is a genuine capability differentiator.

Gemini is available through Google AI Studio for individuals and through Vertex AI for enterprise deployments, which gives you a more managed deployment surface—audit logging, VPC controls, regional data residency—if you need that for compliance reasons.

Licensing: API access, proprietary weights. Google's terms vary between the AI Studio and Vertex AI tiers. The Vertex AI terms are more enterprise-friendly but require a Google Cloud relationship.

Tier Two: Open-Weight Models

Meta Llama

The Llama 3.1 release in July 2024 was a genuine inflection point in this tier. Llama 3.1 405B—at 405 billion parameters, the largest publicly available open-weight model at the time—posted benchmark numbers on MMLU, HumanEval, and GPQA that sat inside the range of GPT-4-class proprietary models. Not at the top of that range. But inside it. That hadn't happened before with a freely downloadable model.

The 128,000-token context window on the 405B and 70B variants was also real progress. The 8B variant, which most developers actually deploy (because 405B requires significant GPU memory—around 800 GB in full precision), has an 8,192-token context limit, which is fine for many tasks and limiting for others. Always check the variant, not just the family name.

Meta's license for Llama 3.x is not the MIT License. It's a Meta-specific license that permits commercial use but includes a restriction: if your product or service has more than 700 million monthly active users, you need to contact Meta for a separate license. For almost everyone reading this book, that restriction is irrelevant. For a handful of companies, it's not.

By mid-2026, Llama 4 variants had entered public availability. The early releases are multimodal, with a vision encoder integrated directly into the architecture rather than added post-hoc. Benchmark performance has continued climbing. The trajectory for Llama has been relentlessly upward.

Mistral

Mistral AI, the French company founded in 2023 by former researchers from DeepMind and Meta, built a reputation early on for producing models that punched well above their parameter count. Mixtral 8x7B introduced many developers to mixture-of-experts architecture for open-weight models: rather than activating all parameters for every token, the model routes each token to a subset of expert feed-forward layers. Active parameter count during inference is lower than total parameter count. The result is a model that runs faster and cheaper than its nominal size suggests, at roughly comparable quality.

Mistral Large 2, released in 2024, handles 128,000 tokens and supports a long list of languages with genuine quality—European languages especially, which isn't surprising given the company's origins. Codestral, Mistral's code-specific model, benchmarks competitively with much larger general-purpose models on code completion tasks. It is a good example of tier-three logic applied inside a tier-two company: a specialized model that wins by being good at one thing.

Mistral's licensing has been more open than Meta's for most of their model line. Several models are available under Apache 2.0, which means genuine commercial freedom. This licensing posture has made Mistral a popular choice for developers who want to run their own stack without legal ambiguity.

Qwen

Alibaba's Qwen team has produced one of the most quietly impressive open-weight families in the landscape. The Qwen 2.5 series—spanning sizes from 0.5B to 72B—shows strong multilingual performance, particularly for Chinese, Japanese, Korean, and other non-Latin languages where most Western model families have historically been weaker.

Qwen 2.5-72B competes with Llama 3.1 70B on most English benchmarks and outperforms it substantially on Chinese-language tasks. If your application serves non-English speakers, this is not a minor point. It's often the decision criterion.

QwQ, Qwen's reasoning-focused variant, generated significant attention in late 2024 as one of the first open-weight models to demonstrate o1-comparable chain-of-thought reasoning on certain math and science benchmarks. The approach is similar to what chapter eleven describes: longer inference, structured intermediate reasoning, budget-controlled generation. The open-weight availability of a capable reasoning model was notable.

Qwen models are primarily licensed under Qwen-specific terms that allow commercial use below certain monthly active user thresholds, similar to Llama's structure. The weights are downloadable from Hugging Face. Running a 72B model at comfortable speed requires A100 or H100 class hardware; quantized versions (GGUF format, available through llama.cpp and similar) run on much more modest setups.

DeepSeek

DeepSeek, the Chinese AI lab backed by quantitative hedge fund High-Flyer, released DeepSeek-R1 in January 2025 and briefly broke the internet. Not metaphorically. Nvidia's stock dropped seven percent in a single day partly in response to the implication that frontier reasoning capability was achievable at dramatically lower training cost.

The story behind R1 is worth understanding. DeepSeek used a training approach based on large-scale reinforcement learning—letting the model discover reasoning strategies through trial and error on math and code problems, rather than distilling from a teacher model—and produced benchmark scores on AIME 2024 and MATH-500 that matched or exceeded OpenAI's o1. The model weights were released under an MIT license for the distilled variants (smaller models trained to imitate the larger R1's reasoning behavior). The full R1 weights are also available, under a separate but still permissive license.

DeepSeek-V3, the base model underlying R1, was trained at reported cost dramatically below comparable Western models. The exact number—reportedly around six million dollars in compute—has been debated and caveated by people who know the hardware market well. But even if the true cost was two or five times that figure, it's still a fraction of what frontier training runs have cost at Anthropic or OpenAI. The efficiency techniques—mixture-of-experts architecture, a custom attention variant called Multi-head Latent Attention, and aggressive hardware-level optimizations—are documented in DeepSeek's public technical reports.

The geopolitical dimension of DeepSeek is real and you should know it exists. There are ongoing regulatory discussions in multiple countries about whether models from Chinese labs should be deployed in sensitive government or enterprise contexts. That's not a technical argument; it's a policy argument. But it affects procurement decisions, and you'll encounter it.

Tier Three: Specialized Fine-Tunes

The open-weight ecosystem has produced a long tail of specialized models that rarely appear on general leaderboards and frequently outperform frontier models on narrow tasks. Understanding this tier is a competitive advantage.

Code is the most developed specialization. DeepSeek-Coder-V2 benchmarks at or above GPT-4o on HumanEval and similar code benchmarks—on a significantly smaller parameter footprint. Codestral from Mistral, as noted, does the same. StarCoder2, produced by the BigCode project through Hugging Face and ServiceNow, is trained on a curated corpus of permissively licensed code and is genuinely production-deployable in contexts where training data provenance matters for legal reasons. These aren't hobbyist projects. They're purpose-built tools.

Medical fine-tunes built on Llama and Qwen base models exist in significant numbers on Hugging Face. Some are research artifacts. A smaller number have been evaluated with actual clinical rigor. The difference between "fine-tuned on medical textbooks" and "validated on clinical decision-making benchmarks with domain expert review" is enormous, and the model card often buries this distinction. Read the model card. Really read it.

Legal, financial, and customer support fine-tunes follow similar patterns. The ones that actually work are trained on domain-specific data by teams that understand the domain, evaluated against domain-specific metrics, and maintained across model updates. The ones that don't work are general models with a system prompt that says "you are a legal assistant." The line between those two categories is wider than it looks from the outside.

The LoRA ecosystem—parameter-efficient fine-tuning adapters that can be loaded on top of base weights at runtime—has made specialized adaptation dramatically cheaper. You don't need to retrain an entire 70B model to adapt it to your company's documentation style or product vocabulary. You train a small adapter on your data, load it alongside the base weights, and get behavior that reflects your domain. The quality ceiling is lower than full fine-tuning, but the cost is orders of magnitude lower, which makes it the right choice for many production use cases.

A Comparative Framework

Rather than a table that will be wrong in six months, here are the dimensions that matter for evaluation, and what each tier generally delivers.

Context length. Proprietary models lead here, with Google's Gemini line offering the longest production context at 1M tokens as of 2026. Open-weight models have caught up significantly—128K is common at the 70B tier—but the million-token range is still largely a proprietary story. If your application genuinely needs to process a full codebase or a year of financial documents in one pass, your options narrow quickly.

Multimodal capability. Proprietary models again. GPT-4o and Gemini handle image, audio, and video inputs in ways that open-weight models are only beginning to match. Llama 4's vision integration is promising but early. If you need robust image understanding at scale today, you're probably calling a proprietary API.

Licensing and deployment flexibility. Open-weight models win decisively. Running Qwen 2.5-72B on your own infrastructure means no per-token cost, no rate limits, no data leaving your environment, and no vendor relationship. For high-volume applications, for privacy-sensitive data, or for contexts where you need to control the entire stack, this is not a preference—it's a requirement.

Cost at scale. This depends so heavily on your usage pattern that any general claim is misleading. A rough heuristic: if you're below about 10 million tokens per day, the operational overhead of self-hosting open-weight models probably isn't worth the API cost savings. Above that threshold, the math starts shifting. Above 100 million tokens per day, self-hosting becomes an obvious economic choice for most use cases.

Reasoning capability. The o-series from OpenAI still leads on the hardest multi-step reasoning benchmarks, with Claude's extended thinking mode close behind. DeepSeek-R1 and QwQ have demonstrated that the gap between proprietary and open-weight reasoning is closeable—the distance has compressed faster than most people expected.

Choosing a Model Is an Engineering Decision

Here is the mistake that gets made constantly, including by experienced engineers: treating model selection as a prestige decision. The question "should we use Claude or GPT-4?" asked without any reference to the specific task, the required latency, the data privacy constraints, the expected token volume, or the acceptable failure modes is not an engineering question. It's a brand question.

Start from the task. What does success look like? Write that down. Then build the smallest possible eval that lets you measure it—twenty representative inputs with labeled correct outputs is enough to start. Run the top two or three candidate models against that eval. The answer you get will be more informative than any leaderboard.

You will frequently discover that a smaller, cheaper model performs within measurement error of the frontier model on your specific task. This happens because frontier benchmarks are designed to stress-test general capability across a wide distribution of tasks. Your task is not that distribution. It's one narrow slice, and a well-chosen smaller model is often fully sufficient for one narrow slice.

You will also occasionally discover that the frontier model is genuinely necessary—because your task requires synthesis across a very long context, or because it requires reasoning depth that smaller models

demonstrably lack. That's also useful information. The point is that you discovered it empirically, not by assumption.

The Honest Shelf Life of This Chapter

Here is what I'm confident will be true regardless of when you read this: the proprietary tier will have released at least one new major model version. The open-weight tier will have at least one new entrant or significant upgrade that the field is arguing about. At least one specialized fine-tune will have gone quietly viral in some developer community for being unexpectedly good at something.

The three-tier framework will still hold. The core trade-offs—deployment flexibility versus capability ceiling, cost versus context length, licensing freedom versus frontier access—will still be the real axes of decision. The specific numbers will have shifted.

This is not a problem to be solved. It's a feature of a technology that is still in rapid development. What you need is not a memorized list of current specs—those are one web search away and always more current than any book. What you need is the framework to evaluate whatever you find when you get there.

That's what this chapter is for.

CHAPTER XVI

Where This Is Going Next (and Where It Isn't)

Predictions about AI age poorly. Sometimes within weeks. So let me be direct about what this chapter is and isn't: it's a snapshot of credible trajectories as of 2026, grounded in published research and observable patterns, not a forecast dressed up as certainty. The field moves fast enough that any specific number here carries an expiration date. What won't expire are the underlying mechanisms — and if you understand those, you can evaluate the next wave of announcements yourself without needing someone to tell you what to believe.

With that established, let's look at where the evidence actually points.

Context Gets Longer. The Problem Is Making It Useful.

The raw context window numbers have climbed steeply and show no signs of plateauing. GPT-4 launched in March 2023 with an 8,192-token context. By the end of that year there was a 128K variant. Claude 3 shipped with 200,000 tokens. Google's Gemini 1.5 Pro demonstrated a 1,000,000-token context in its February 2024 technical report — processing an entire codebase, or the full text of roughly a dozen novels, in a single forward pass.

That trajectory continues. Architectural improvements — ring attention, sparse attention variants, and continued hardware advances — are making million-token-plus windows cheaper to operate. Training on longer sequences is getting more tractable as distributed training infrastructure matures.

Here's the thing nobody should let you gloss over: a long context window is a capability, not a solution. Liu et al.'s 2023 paper "Lost in the Middle" showed that models perform significantly worse on retrieval tasks when the relevant information is buried in the middle of a long context rather than positioned near the beginning or end. The attention mechanism, in practice, doesn't distribute uniform weight across 100,000 tokens. Information near the edges of the sequence gets attended to more reliably. Put the critical fact at token 50,000 of a 100,000-token context, and the model is measurably more likely to miss it than if you'd put it at token 1,000.

This is not a quirk that will simply be trained away, because it reflects something fundamental about how positional encoding and attention interact at scale. It's improving — newer models show better mid-context retrieval than their predecessors — but "improving" and "solved" are different things.

The practical trajectory here is twofold. First, raw window length will keep increasing. Second, and more importantly, the tooling around how you use that window — smart chunking, context compression, selective retrieval that inserts only the relevant portions — will matter as much as the size itself. Retrieval-augmented generation isn't a workaround for small context windows; it's a first-class architecture pattern that remains

useful even when context is enormous. The reason is attention budget. More tokens in the window means the model's attention is diluted across more candidates. You still benefit from curating what goes in.

Inference Gets Cheaper. Fast.

The cost of running LLM inference has dropped so fast that the numbers from two years ago look almost satirical in retrospect. In 2023, calling GPT-4 cost roughly $0.03 per 1,000 input tokens and $0.06 per 1,000 output tokens. By early 2025, GPT-4o-mini — a smaller, distilled model competitive with earlier GPT-4 on many tasks — cost $0.00015 per 1,000 input tokens. That's a factor of 200 reduction in per-token cost in under two years.

Two technical forces drive this, and both will keep pushing costs down.

Quantization is the practice of reducing the numerical precision used to store and compute model weights. A standard 32-bit floating-point representation can be compressed to 16-bit, then to 8-bit (INT8), then to 4-bit (INT4) with acceptable quality degradation on most tasks. A 70-billion-parameter model stored in 16-bit takes roughly 140GB of GPU memory. The same model in 4-bit quantized form fits in about 40GB — close enough to run on a single high-end GPU rather than a multi-GPU server. Formats like GGUF (used by llama.cpp) and GPTQ have made it possible to run Llama 3.1 70B on a Mac Studio with 192GB unified memory. On a laptop. This is not speculative. People are doing it today.

Distillation is the practice of training a smaller model to mimic the output distribution of a larger one. The smaller model — the "student" — is trained not just on raw data but on the token probabilities the larger "teacher" produces, which gives the student richer supervisory signal than one-hot ground truth labels. DeepSeek's work in 2024 and 2025 demonstrated distilled models that matched or approached frontier performance on coding and math benchmarks at a fraction of the parameter count. Anthropic and OpenAI have both released smaller, cheaper model variants clearly shaped by the same dynamic.

The practical implication for you, building on top of these APIs: the economic case for "just use the biggest model for everything" is eroding every quarter. A well-scoped classification task, a structured extraction job, a code completion autocomplete — these don't require a 1,000-billion-parameter frontier model. They require a model good enough to clear your accuracy threshold at a latency your users will tolerate. Chapter 13 covered the engineering logic of model selection. The trend being described here is that the gap between "big model" capability and "small model" capability is compressing, which means the right threshold for reaching for the big model keeps getting more demanding to justify.

This trajectory is durable. Quantization research isn't plateauing. Distillation techniques are improving. Hardware — the H100, the H200, the next generation of ASICs from Google, Amazon, and dedicated AI chip startups — keeps increasing throughput per dollar. Inference will be substantially cheaper in 2027 than it is today.

Open-Weight Models Close the Gap

For most of 2023, the capability gap between proprietary frontier models and the best publicly available open-weight models was wide. GPT-4 performed measurably better than anything you could download and run yourself on most standard benchmarks. That gap has compressed substantially, and the compression is accelerating.

Meta's Llama 3.1 405B, released in July 2024, matched GPT-4 on several standard benchmarks including MMLU and HumanEval. It's a model you can download, fine-tune, run on your own infrastructure, and modify without asking anyone's permission. Mistral's models consistently punched above their weight class relative to parameter count. Alibaba's Qwen 2.5 series, released in late 2024, showed strong multilingual performance and coding capability. DeepSeek-R1, released in January 2025, demonstrated reasoning performance competitive with OpenAI's o1 on math and logic tasks — and was released as an open-weight model with permissive licensing.

The dynamic here matters for how you think about the ecosystem. Proprietary labs have to build enormous moats: exclusivity, API convenience, multimodal capability, safety tuning, enterprise support. Open-weight models have a different set of advantages: you can fine-tune them on proprietary data without sending that data to a third-party API, you can run them on-premise, you can control the exact version and behavior, and you pay for compute rather than per-token API fees at scale.

The credible trajectory: open-weight capability will keep improving. The gap between what you can run yourself and what you can get from the most advanced proprietary API will likely remain measurable — proprietary labs have resources that even well-funded open-source efforts can't fully replicate — but it will not be the decisive gap it once was. For many production tasks, an open-weight model running on your own infrastructure will be the right answer by the time you're reading this.

What won't change is the training data and compute advantage at the very frontier. Training a model the size of GPT-4 or Gemini Ultra requires compute budgets measured in hundreds of millions of dollars. That limits the frontier. But "frontier" and "fit for purpose" are not synonyms.

Tool Use and Agents Become Native, Not Bolted On

Early function calling in LLMs was an overlay. You prompted the model with a description of available tools. The model, trained on a mix of general text, learned to produce something that looked like a function call if you set things up correctly. It worked, mostly, but it was a parlor trick coaxed from a base capability rather than something the model was explicitly built for.

That is changing. Newer models are trained with tool use as a first-class objective. The instruction tuning datasets include explicit examples of tool call sequences, multi-turn tool-use trajectories, error recovery from malformed tool responses, and reasoning about which tool to use and when not to use any of them. The result is qualitatively different — not just more reliable structured output, but better judgment about tool selection and more graceful handling of tool failures.

The agent-native direction extends this further. Rather than a model that responds once and returns, you get architectures where the model operates in loops: plan, act, observe, revise. OpenAI's work on its agentic products, Anthropic's Computer Use (released in late 2024), and the broader ecosystem around frameworks like LangGraph and multi-agent orchestration libraries all reflect the same underlying direction. The model isn't changing fundamentally — it's still a next-token predictor — but the scaffolding around it, and the training that teaches it to work within that scaffolding, is being built deliberately rather than improvised.

One clarification worth holding onto from Chapter 10: the model still doesn't execute anything. It still writes. The runtime executes. What's improving is the model's ability to write the right things at the right time in the right format, and to reason about the results that come back. The distinction matters because it tells you where failures live. If a tool call fails, debug the runtime and the interface before you blame the model. If the model chooses the wrong tool, or misinterprets a tool result, that's a model behavior — addressable through fine-tuning or prompting.

The near-term trajectory: agents will become more reliable and longer-running. Tasks that currently require human checkpoints every few steps will run further autonomously before needing intervention. This will not happen uniformly. Long-horizon tasks with ambiguous intermediate states will remain hard. The failure mode — an agent confidently proceeding down a wrong branch for dozens of steps before the error compounds into something unrecoverable — is a real one that current architectures handle poorly. Reliability at agent scale is an active research problem, not a solved one dressed up in product announcements.

Now the Honest Part

Everything above represents genuine progress. Measurable, documented, replicable progress. The trends described are not speculative; they're extensions of trajectories already clearly established in published research and deployed products.

What follows is the part that gets left out of most "future of AI" writing, because it doesn't generate the right kind of excitement. These are the problems that remain genuinely hard. Not "hard but on a clear path to solution." Hard in the sense that the current paradigm doesn't obviously solve them, and nobody has demonstrated a reliable fix.

Multi-Step Reasoning at Scale

Chain-of-thought prompting, as Wei et al. showed in 2022, measurably improves performance on multi-step reasoning tasks by converting implicit computation into attended context. OpenAI's o-series models and Anthropic's extended thinking approach extend this further, allocating substantially more compute at inference time to generate a long internal scratchpad before producing a final answer.

These approaches work. They measurably improve performance on math competitions, logic puzzles, and structured coding tasks. The ARC-AGI benchmark — designed by François Chollet specifically to resist pattern-matching from training data — showed dramatic improvement from o3 in early 2025. That's real.

What isn't real is the extrapolation that longer chains of thought solve the general reasoning problem. The failure mode of chain-of-thought is subtle and persistent: the model can produce a reasoning chain that looks coherent, follows an internally consistent structure, and arrives at a wrong answer. Not because any individual step is obviously wrong, but because the model has drifted from the actual constraints of the problem while maintaining the surface appearance of rigorous logic. This is sometimes called "reasoning-chain drift" — Chapter 7 covered it in the context of hallucination — and it's endemic, not incidental.

The deeper issue is that the model has no ground truth to check against during generation. It's predicting the next plausible token in a reasoning chain, which means plausible reasoning steps that lead somewhere wrong are just as accessible to the sampler as correct ones. Extended thinking helps because more compute means more opportunities to self-correct, but it doesn't change the fundamental mechanism. There is no internal fact-checker. There is no proof verifier running in parallel. There's just continued next-token prediction over a longer sequence.

Verifiable formal domains — math, code, games with explicit rules — are the places where reasoning models improve most reliably, because there's an external check. You can run the code. You can verify the proof. In open-ended factual or analytical reasoning, you can't. That distinction matters enormously for where you should and shouldn't trust a reasoning model's outputs.

Factuality at Generation Time

Retrieval-augmented generation reduces hallucination. Grounding a model's response to retrieved documents that the model is instructed to use gives the generation a factual anchor. This is a meaningful improvement and deserves the adoption it has gotten.

It does not solve hallucination. Three things that retrieval doesn't fix:

First, the model can still confabulate within or around the retrieved content. If the retrieved passage says "the company was founded in 1987" and the model generates "the company, founded in 1987 by its longtime CEO," the CEO detail just appeared — the retrieval said nothing about it. This happens.

Second, retrieval doesn't help when the relevant fact isn't in the retrieved corpus, or when the query doesn't retrieve the right document. Your retrieval system has its own failure modes — embedding mismatches, stale indexes, coverage gaps — and the language model downstream has no way to know when those failures have occurred. It will generate text that sounds grounded regardless.

Third, the deeper problem isn't just "which facts does the model know." It's that the model has no mechanism, at inference time, to distinguish between high-confidence retrieved knowledge and low-confidence generated plausibility. Both feel the same from inside the generation process, because they are mechanically the same: they're both token sequences being predicted. The model doesn't "feel uncertain" in any structural sense. It produces tokens. Calibrated uncertainty — knowing what you don't know — is a behavior that can be tuned toward but not guaranteed.

The honest status of factuality work as of 2026: researchers know the problem well. There are partial mitigations. There is no durable solution on the horizon that works across open-ended domains. This is not a reason to avoid using LLMs for factual tasks; it's a reason to build your systems with explicit verification steps, to keep humans in the loop for high-stakes factual claims, and to treat every unsourced LLM output as requiring confirmation rather than as a finished fact.

Alignment at Scale

Anthropic's Constitutional AI approach and OpenAI's RLHF-based alignment work represent serious, well-funded, carefully documented efforts to build models that behave well: that are helpful, honest, and don't cause harm. These efforts work well enough that you can hand current models to millions of users and not produce catastrophic outcomes at the frequency you might fear. That's a genuine achievement.

What remains genuinely unsolved is alignment at scale — specifically, the problem of ensuring consistent behavior across the distribution of inputs a deployed model will actually encounter, not just the distribution of inputs it was evaluated on during training.

The core difficulty: RLHF optimizes human rater preference on sampled outputs during training. Human raters see a finite sample of possible inputs. The space of possible inputs to a deployed model is enormous and includes many that no rater ever saw. Behavior in-distribution, on inputs similar to those seen during training, tends to be well-aligned. Behavior far from that distribution is less predictable — and users are creative, persistent, and numerous enough to find the edges.

There's also the problem of what happens as models become more capable. A relatively less capable model trying to deceive you is easy to catch. A substantially more capable model, if it were misaligned, is harder to catch — it can produce outputs that satisfy alignment evaluations without being aligned in the way the evaluations intend to measure. This is not a theoretical concern about fictional superintelligence. It's a concrete empirical problem that becomes harder as the models under evaluation become better at prediction and language modeling. You cannot evaluate for deception using a test the system under evaluation can pass by modeling what the test expects.

Anthropic's interpretability research — the Superposition paper, mechanistic interpretability work trying to understand what circuits inside a transformer actually compute — is an honest attempt to build tools for catching these failures. It's promising. It is also early and narrow in scope. We can identify some circuits. We cannot read the "intent" of a model in any rigorous sense. The field doesn't have the tools yet.

This doesn't mean alignment work is hopeless. It means calibrated skepticism is warranted when any lab — including the good ones — claims their model is reliably safe. "Safe enough for most users on most tasks, with monitoring" is an honest statement. "Robustly aligned" is not.

What This Means for You, Building Now

The honest summary: the near-term is genuinely exciting. Inference gets cheaper. Context gets longer and more useful. Open-weight models give you more options and more control. Tool use and agents become more reliable. These are not hype — they're documented trends extrapolated from real evidence.

The hard problems remain hard. Multi-step reasoning over long chains is better but not reliable. Factuality at generation time is mitigated but not solved. Alignment at scale is an open research problem with no clear solution in sight.

What this argues for, practically, is an engineering posture rather than a faith posture. Build systems that assume the model will sometimes be wrong. Build evaluation frameworks that tell you when and on what it's wrong. Use the cheaper, smaller, faster models when they clear your threshold — and measure that threshold explicitly rather than guessing it. Keep humans in loops where mistakes are expensive. Treat frontier model announcements as starting points for your own evaluation, not as conclusions.

The researchers building these systems are not hiding the problems from you. The limitations described in this chapter appear, with appropriate technical rigor, in published papers from OpenAI, Anthropic, Google, Meta, and academic groups worldwide. What gets lost in the translation from research to product announcement is not usually a lie — it's the gap between "demonstrated in controlled evaluation" and "reliably true in your specific production environment." That gap is yours to measure.

One last thing. The history of this field has been repeatedly surprised in both directions — capabilities arrived faster than expected, certain problems proved more stubborn than expected. The people confidently predicting where this is going in five years are, at best, making informed guesses. The people telling you it's all about to converge into general intelligence on a three-year timeline and the people telling you it's all hype that will crash into a wall are both more certain than the evidence supports.

What you can do — what this book has tried to give you the tools to do — is read the primary sources, understand the actual mechanisms, and evaluate claims on their merits. That turns out to be more valuable than any particular prediction about where the field lands. The mechanisms don't change when the marketing does.