A GROUND-UP FIELD GUIDE

How an LLM Works

From a single token to an agent that acts

The giant kingfisher (Megaceryle maxima) is the largest kingfisher in Africa, measuring roughly 42 to 46 cm (17 to 18 in) in length and weighing between 255 and 425 g. It is a resident breeder across most of sub-Saharan Africa, recognized by its large, dagger-like black bill, shaggy crest, and distinctive black-and-white mottled plumage.

This text was extracted from the internet by an AI — a Large Language Model — and this book is an attempt to give the reader an intuitive grasp of how these LLMs work, from the ground up.

BY

Dr Neal Aggarwal

drnealaggarwal.info

A GROUND-UP ACCOUNT

How an LLM Works

From a single token to an agent that acts

Dr Neal Aggarwal

drnealaggarwal.info · eight chapters, built from first principles

Contents

CHAPTER 1

The Grain of Language

How a Machine Reads the Internet

↑ Back to Contents

Type a sentence into ChatGPT and press enter. In the fraction of a second before any answer begins to form, something happens that feels too mundane to matter and turns out to explain a startling amount about what these systems can and cannot do.

Your sentence is taken apart.

Not into words, exactly, and not into letters. Into pieces of a kind you have almost certainly never thought about — pieces the machine invented for itself before it ever met your sentence. Everything the model does afterward — every association it draws, every fact it recalls, every mistake it makes — happens downstream of this quiet act of disassembly. If you want to understand large language models from the ground up, and I mean genuinely from the ground, this is the ground. So this is where we start.

I have spent a long time teaching difficult technical material to people who are clever but not specialists — clinicians, mostly, and researchers from fields far from computer science. The lesson that experience has burned into me is that understanding almost always fails at the *foundation*, not the summit. People nod along at "it predicts the next word" and "it's a neural network," and then a hundred pages later nothing quite adds up, because the foundation was skipped. Tokenization is the most-skipped foundation of them all. Books gesture at it in a paragraph and move on. We are not going to move on. By the end of this chapter you will understand it better than most people who use these tools professionally, and you will never look at a chatbot's odd little failures the same way again.

Part I — Where the words come from

Before we can talk about how a model reads, we have to talk about what it reads. And the honest answer, stripped of mystique, is: a very large pile of text, most of it scraped from the open internet.

This sounds almost disappointingly ordinary, and it is worth sitting with the strangeness underneath the ordinariness. A modern language model's entire sense of the world — its grasp of grammar, its store of facts, its imitation of reasoning, its uncanny fluency in dozens of languages — is a statistical residue left behind by an enormous quantity of human writing. There is no encyclopaedia inside it, no database, no rules typed in by hand. There is only text, and what the model managed to internalise by trying, billions of times, to predict what came next in that text. Get the text wrong and everything downstream is wrong. The corpus is not a detail of the build; it *is* the build.

So where does the pile come from? The starting point, for most of the field, is a remarkable public resource: Common Crawl, a non-profit that has been quietly sending automated crawlers across the web since 2007 and archiving what they find. The scale is difficult to hold in the mind — over 300 billion pages accumulated across fifteen years, with three to five billion fresh pages added every month. It is, in a real sense, a photograph of the readable internet, taken over and over.

But here is the thing nobody tells you: that raw photograph is almost unusable. The open web is, to put it kindly, mostly junk. Spam, boilerplate navigation menus, cookie banners, auto-generated filler, adult content, malware pages, duplicated text copied across ten thousand sites, and markup — endless HTML tags wrapped around the actual prose like packaging around a small gift. Feed that directly to a model and you get a model that has learned the internet's worst habits. The single least glamorous and most decisive part of building a language model is *cleaning*.

The cleaning is a pipeline — a sequence of filters, each throwing away more of the raw crawl until what remains is something you would actually want a model to learn from.

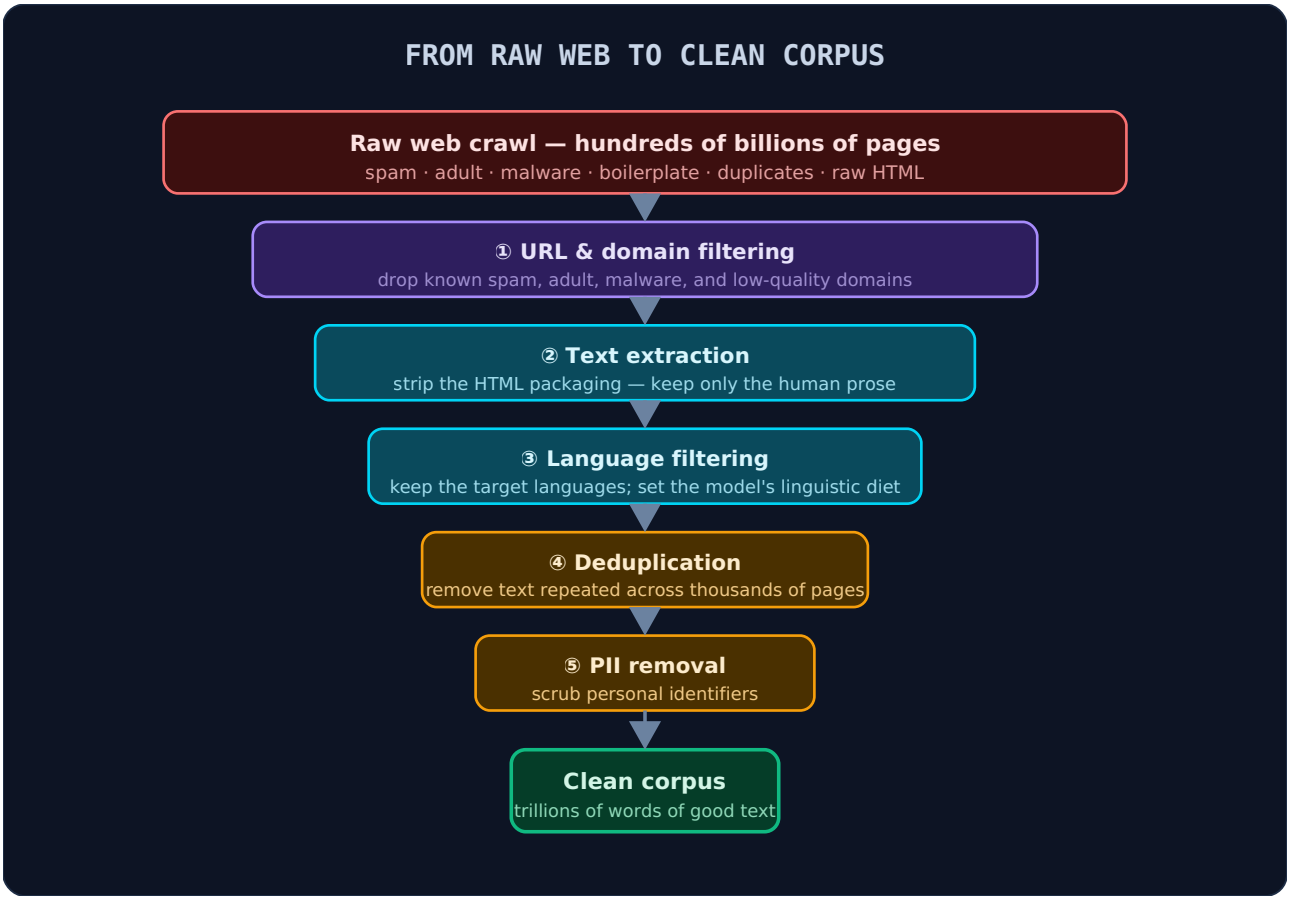

Figure 1. The refinement pipeline. Each stage discards more of the raw crawl. What survives — a few tens of terabytes of clean text from a vastly larger input — is the material the model actually learns from. Two of these stages, text extraction and language filtering, quietly decide what the model will be good at.

If you would like to see how meticulous this cleaning actually is, the team at Hugging Face documented one such pipeline in unusual and generous detail for a dataset they call FineWeb — around fifteen trillion words of filtered English text, occupying roughly forty-four terabytes on disk. Read their write-up and one point becomes inescapable: an

enormous fraction of the effort in building a frontier model goes not into the clever mathematics you will meet in later chapters, but into deciding, page by page and filter by filter, what counts as text worth learning from. It is the most consequential editorial decision in the field, and it is made almost entirely by engineering.

> **A note for those of us in medicine, and it applies far beyond it.** Look again at stage ③, language filtering. A pipeline tuned to keep English and discard "low-resource" languages is, in the same stroke, deciding that the model will be fluent in the medicine of the English-speaking world and clumsy in Swahili, in Amharic, in the clinical vocabulary of the places that most need cheap expertise. The model's blind spots are not accidents of the algorithm. They are inherited, faithfully, from what we chose to feed it. Every limitation you will later curse in a deployed tool was, somewhere, a line in a data-cleaning script.

So: we now have our pile. Tens of terabytes of clean, deduplicated, mostly-English human writing. A model cannot read a terabyte of text any more than it can read your sentence. It reads numbers. Which brings us to the real subject of this chapter — the bridge between text and number, and the surprising cleverness required to build it well.

Part II — Why a machine cannot simply read

Here is a fact that is easy to state and easy to underestimate: a neural network does not process text. It processes sequences of numbers drawn from a fixed, finite set of symbols. Everything a model ever "sees" is a one-dimensional stream of these symbols, one after another, like beads on a string. Our entire job, in this chapter, is to turn a page of writing into such a stream — and to do it well, because *how* we do it turns out to matter enormously.

Let us take the most naive approaches first, because their failures are exactly what motivates the clever solution.

Attempt one: one symbol per character. The obvious idea. Let each distinct character — every letter, digit, punctuation mark — be a symbol. "cat" becomes three symbols: `c`, `a`, `t`. Clean and intuitive. But now consider that the model must handle not just English but every script humans write in: Cyrillic, Arabic, Chinese, Japanese, Korean, mathematical notation, emoji, and the thousands of rarer symbols in between. The universal catalogue of these characters, called Unicode, contains around 150,000 distinct code points and grows every year. A vocabulary of 150,000 symbols is workable in principle, but it is unstable — tied to a standard that keeps changing — and it wastes enormous capacity on symbols the model will almost never see. Worse, it still splits common English words into many symbols, which, as we are about to discover, is costly.

Attempt two: work in raw bytes. There is a beautiful, principled alternative. Every piece of digital text, in whatever script, is *already* stored as a sequence of bytes, using an encoding called UTF-8. A byte is just a number from 0 to 255 — so there are exactly 256 possible byte-symbols, a tiny and permanently fixed vocabulary. UTF-8 spends one byte on common English characters and two, three, or four bytes on rarer ones. It is

universal, stable, and elegant. Let each byte be a symbol, and the vocabulary problem vanishes.

To see the ladder of representations underneath a single ordinary word, look closely:

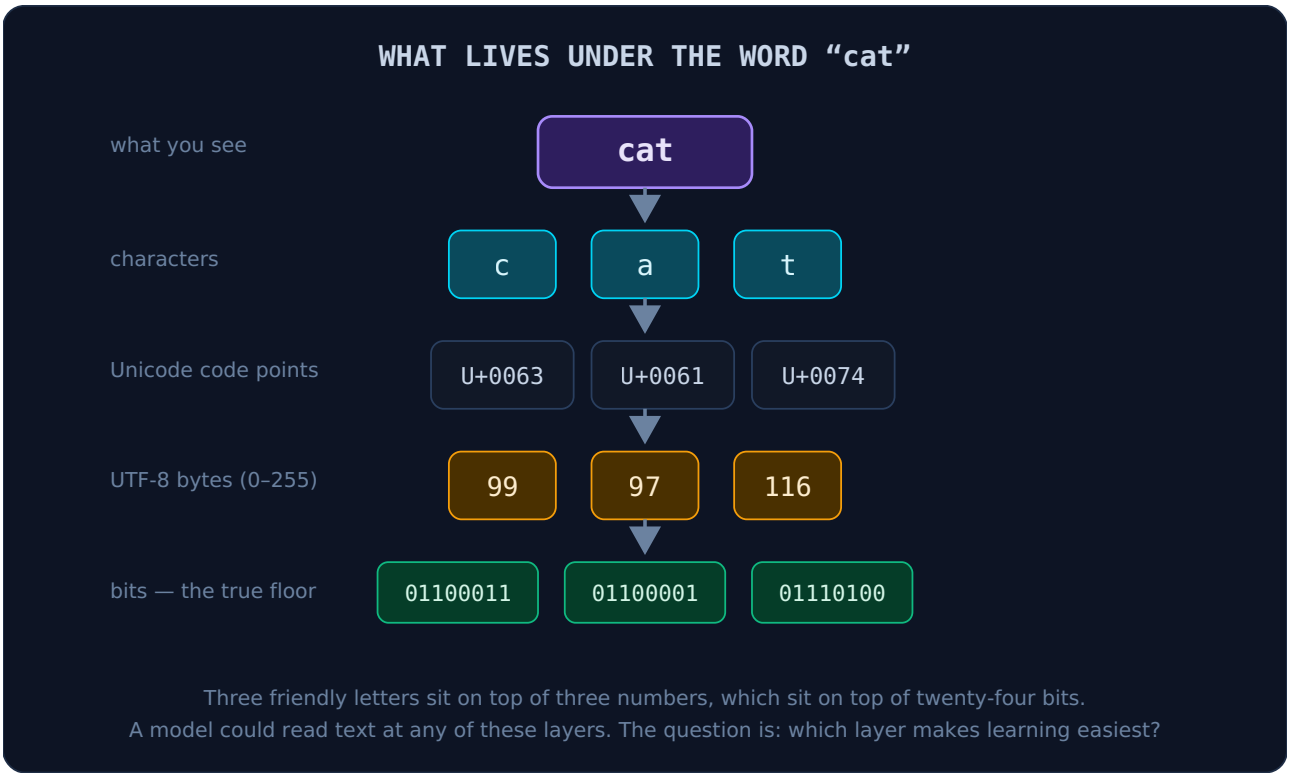

Figure 2. Every character rests on a stack of representations. Working in raw bytes (the amber row) gives a permanently fixed vocabulary of just 256 symbols. It is tempting to stop here — but there is a hidden cost, and it is the cost that forces the whole field toward something cleverer.

The hidden cost of bytes is *length*. In UTF-8, an ordinary English word is several bytes long, and a paragraph is thousands. If every byte is a symbol, then the sequence the model must chew through becomes very long indeed. And here we run into a fact that will echo through every later chapter of this book: the machinery that lets a model relate each symbol to every other symbol grows *quadratically* with sequence length. Double the number of symbols and you roughly quadruple the work. Sequence length is the single most expensive resource a language model has. Spending it one byte at a time is ruinous.

So we are caught between two pressures, and naming them precisely is the key to everything that follows.

> **The central tension of tokenization.** A *small* vocabulary (like 256 bytes) is simple and universal but produces *long* sequences, which are slow and expensive. A *large* vocabulary produces *short* sequences but demands more of the model and leaves rare symbols starved of examples. We do not want either extreme. We want a vocabulary sitting in a carefully chosen middle — large enough to keep sequences short, small enough to stay learnable. The art of tokenization is finding that middle. And, wonderfully, we do not choose the middle by hand. We let the data choose it.

Figure 3. Every tokenizer is a chosen point on this seesaw. Modern models land near a vocabulary of one to two hundred thousand symbols — the empirically-discovered sweet spot where sequences are short enough to be affordable and the vocabulary is small enough that every symbol still appears often enough to be learned well.

Part III — The machine builds its own alphabet

Now comes the idea at the heart of this chapter, and it is genuinely lovely. We are not going to hand the model a vocabulary. We are going to let it *grow* one from the data, greedily, by noticing which combinations of symbols happen most often and promoting them to symbols in their own right.

The method is called **byte pair encoding**, or BPE, and once you see it you cannot unsee it. Start from the humble byte vocabulary — 256 symbols, one per possible byte. Now run through the entire clean corpus and ask a single question: *which pair of adjacent symbols occurs most frequently?* In English text the answer, early on, is something like the pair `t` followed by `h`. So we do the obvious, audacious thing: we invent a brand-new symbol — call it token number 256 — that *means* "th", and we replace every adjacent `t`, `h` in the corpus with this single new token. The corpus just got a little shorter, and our vocabulary just got one symbol larger.

Then we do it again. With "th" now a single symbol, perhaps the most common adjacent pair becomes "th" followed by `e` — so we mint token 257 meaning "the". Again the corpus shrinks; again the vocabulary grows. And again, and again, tens of thousands of times, each new symbol built from two older ones, each merge chosen purely because the data made that pair common.

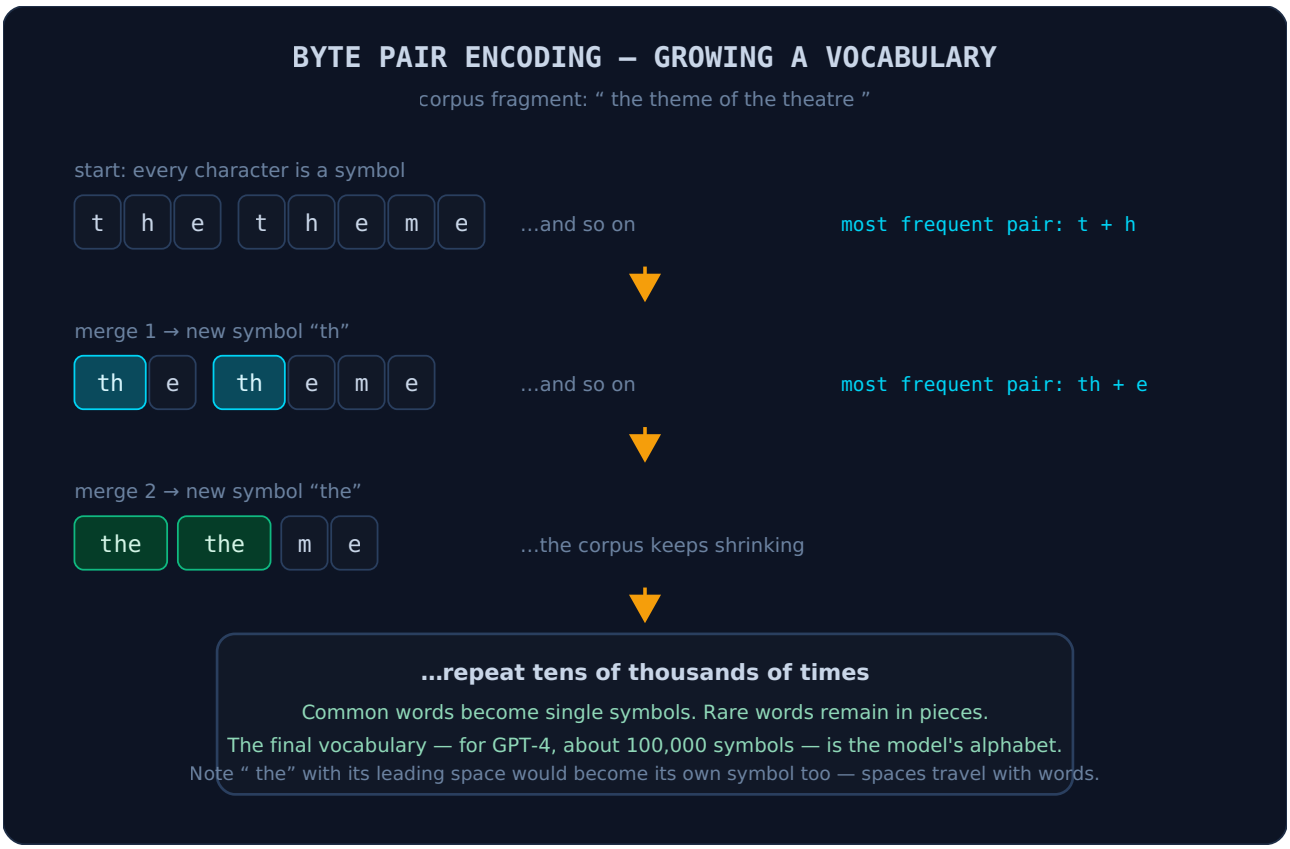

Figure 4. BPE in motion. Notice the direction of travel: we do not chop words up, we *build symbols up* from bytes, guided entirely by frequency. The words you use constantly collapse into single tokens; the words you rarely use stay fragmented. The vocabulary is a fossil record of how the training corpus was actually written.

When this process halts — for GPT-4's tokenizer, at a vocabulary of 100,277 symbols; for its successor GPT-4o, closer to 200,000 — you are left with something remarkable. The most common English words are each a single symbol. Common word-fragments like "ing", "tion", "pre" are single symbols, so even a word the tokenizer has never seen can be assembled from familiar pieces. And genuinely rare strings — an unusual surname, a chemical name, a snippet of code — dissolve into several small tokens, or in the limit, back into individual bytes. Nothing is ever un-representable, because at the very bottom the byte layer can always spell anything out. The scheme is, to use the technical words, *lossless* and *reversible*: you can always reconstruct the exact original text from the tokens, byte for byte.

These learned symbols are what we call **tokens**, and from here on they are the true units of everything. Not words. Not letters. Tokens — the alphabet the machine assembled for itself, one greedy merge at a time.

> **For the tinkerers.** This is not a black box you have to take on faith. OpenAI released the exact tokenizer their models use as an open-source library called tiktoken, and it ships with a small educational module that will walk the BPE merges in front of you, step by step, on any text you give it. If you know a little Python, an hour spent watching it build a vocabulary from a paragraph of your own writing will teach you more than this entire chapter. That is not false modesty; it is how I learned it myself.

Part IV — Looking the tokenizer in the eye

Diagrams and prose can only take intuition so far. The thing that finally made tokenization *click* for me — and for nearly everyone I have taught it to — was seeing a real tokenizer chew on real text, live. So I am going to ask you to stop reading for a few minutes and go do exactly that.

The tool I send everyone to is Tiktokenizer. Open it, choose a model such as GPT-4o, and type into the box. As you type, the text is split into coloured tiles — one per token — with a running count and, if you look, the integer ID of each token. Type your name. Type a sentence. Type a paragraph of a clinic letter. Watch where the cuts fall. It is quietly mesmerising, and it will surface a series of facts that no amount of reading prepares you for. Let me walk you through the ones that matter most, because each one is a lesson in disguise.

Lesson one: the boundaries are not where you expect. Type `tokenization` and you may find it is not one token but two or three — perhaps `token` + `ization`. The tokenizer has no notion of "words." It only knows the fragments its merges happened to build. Common words are whole; longer or rarer ones fracture along the seams of frequency, not the seams of meaning.

Lesson two: the space is part of the word. This one surprises everyone. In the tokenizer's eyes, `world` and `world` — the second with a leading space — are *different tokens with different ID numbers*. The space travels attached to the front of the word that follows it. It makes sense once you see it: in real text, words are almost always preceded by a space, so " world" is far more common than a bare "world", and BPE duly gave the spaced version its own symbol. But it means the model's internal notion of a word includes its leading whitespace, which is not how any human thinks about language.

```
the sentence “The patient took two tablets” becomes six tokens →
The  patient  took  two  tablets  ”
every tile is one token · note the leading spaces riding along inside four of them
```

Lesson three: capitalisation makes a different symbol. `Hello`, `hello`, `HELLO`, and `Hello` are, as far as the tokenizer is concerned, four unrelated symbols. The model has to *learn* from scratch that they are related — that the capitalised form at the start of a sentence means the same thing as the lowercase form mid-sentence. Nothing about the tokens tells it so. This is our first glimpse of a recurring theme: the tokenizer hands the model a set of arbitrary symbols and forces it to rediscover, statistically, the relationships that are obvious to us.

Lesson four: numbers are a mess. Type `127 + 677 = 804` and watch what happens to the digits. You will very often find the numbers carved into ragged pieces — `127` might be one token while `677` splits into `67` and `7`, and `804` into `80` and `4`. There is no consistent "one token per digit" or "one token per number" rule; the cuts fall wherever the training frequencies happened to put them. Hold on to this observation. It is, as we

will see in a moment, the reason a system that can write a sonnet can also insist that 9.11 is larger than 9.9.

Lesson five: not all languages are equal. Paste an English sentence and count the tokens. Now paste its translation into a language that was thin in the training data — Swahili, say, or Amharic — and count again. The second will use noticeably more tokens to say the same thing, sometimes two or three times as many. The tokenizer's merges were learned from a corpus dominated by English, so English got the efficient single-token words while other languages were left in smaller, more expensive pieces.

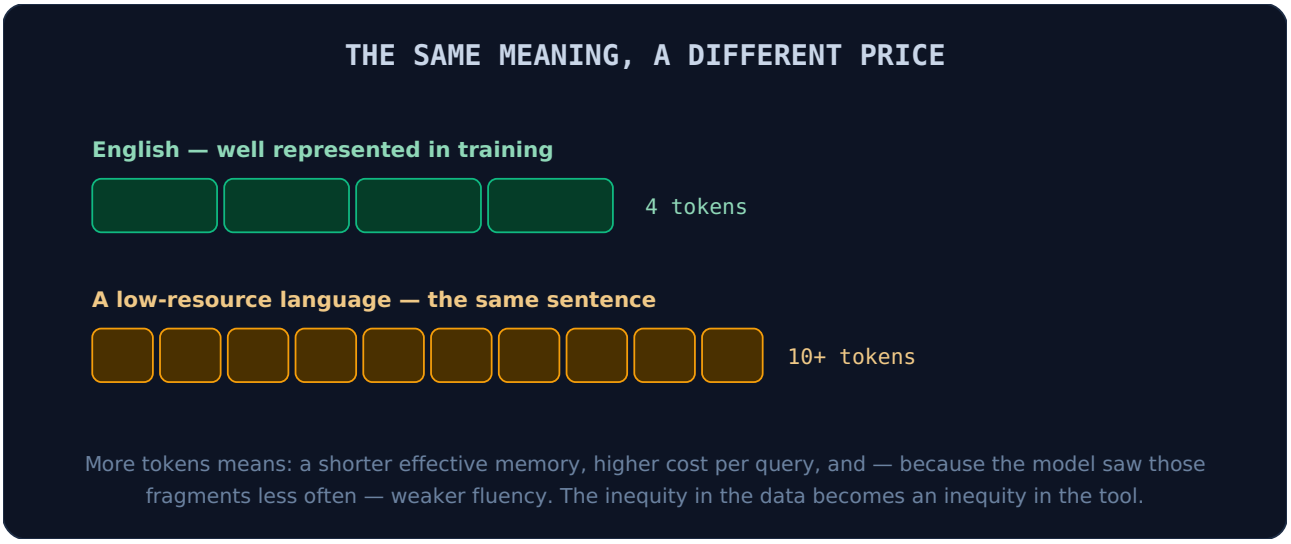

Figure 5. Tokenization efficiency is not a neutral technicality. Because the context window and the price are both measured in tokens, a language that tokenizes inefficiently is, in effect, charged more and remembered less. For anyone hoping to deploy these tools in the world's under-served languages, this is a design constraint to reckon with from day one — not a footnote.

Part V — Why the disassembly explains the machine

We now arrive at the payoff, the reason I insisted we spend a whole chapter on what looks like plumbing. A great many of the notorious, widely-mocked failures of language models are not failures of *intelligence* at all. They are direct, predictable consequences of the fact that the model never sees letters — it sees tokens. Once you hold that fact firmly, a whole category of mystery evaporates.

Consider the famous one: *how many times does the letter "r" appear in "strawberry"?* Models have stumbled on this for years, and every time they do, someone declares that artificial intelligence is a fraud. But look at what we now know. To the model, "strawberry" is not a row of ten letters. It is two or three tokens — perhaps `straw` and `berry`. The individual letters are *welded inside* those tokens, invisible and inaccessible. Asking the model to count the r's is like asking you to count the brushstrokes in a painting you are only allowed to view from across the room. The information has been compressed away at the very first step. The wonder is not that it sometimes miscounts; the wonder is that it manages as often as it does, by having memorised, statistically, facts about its own tokens.

> *The model does not read the way we read. It never meets the letter. It meets the token — and a token is a wall between the machine and the fine grain of the word.*

The same lens clarifies the arithmetic troubles. We saw in Tiktokenizer that numbers fracture into inconsistent pieces. A model asked to add two numbers is therefore not manipulating digits in orderly columns the way a child is taught to; it is pattern-matching over lumpy, irregular tokens whose boundaries bear no relation to place value. That such systems can do *any* reliable arithmetic is a testament to how much structure they extract despite the tokenizer, not because of it. And it is precisely why serious systems now hand arithmetic off to a calculator tool rather than trusting the raw model — a theme we will return to when we discuss tools and agents.

> **The clinical translation, because this is where it stops being a curiosity.** If a model cannot reliably count the letters in "strawberry," think hard before trusting it to notice that "hydralazine" and "hydroxyzine" differ by a few characters, or to catch a transposed digit in a dose. The failure mode is not random; it is structural, and it lives at the tokenizer. This does not make the tools useless — it makes them tools with a known blind spot, to be wrapped in checks precisely where that blind spot bites. Knowing *why* the blind spot exists is the difference between deploying carefully and deploying blindly.

There is even a genre of outright bizarre behaviour that traces to tokenization: rare tokens that were minted during the vocabulary-building phase but then barely appeared in the actual training text. These "glitch tokens" — famous examples were dug out of early GPT models — are symbols the model possesses but was never taught to use, and prompting with them can produce strange, evasive, or nonsensical output, like pressing a key on a piano that was installed but never tuned. It is a small, eerie reminder that the vocabulary and the training are two separate steps, and that a symbol's mere existence does not guarantee the model knows what to do with it.

Part VI — From tokens to the machine

Let us close by placing this chapter in the arc of the book. We began with the open internet and refined it, filter by filter, into a clean corpus. We then confronted the problem that a neural network reads only sequences of symbols, weighed the naive options, and arrived at byte pair encoding — the method by which the machine grows its own alphabet of tokens, balancing vocabulary size against sequence length. We looked a real tokenizer in the eye, and we saw how its quirks ripple outward into the model's most-mocked failures.

Everything from here on operates on tokens. When you read, in later chapters, that a model has a "context window" of, say, 128,000 — that number is counted in *tokens*, not words or characters, which is why the efficiency of tokenization directly determines how much real text the model can hold in mind at once. When we build the prediction machinery, its input will be a sequence of token IDs and its output will be a probability spread across the whole token vocabulary — "given these tokens, which token comes

next?" The tokenizer is the lens through which the model views all of language, and, as with any lens, its particular distortions shape everything seen through it.

If you would like a preview of where these tokens are headed — the vast machine they are about to enter — there is a beautiful interactive visualiser that renders a working language model in three dimensions and lets you walk through it, stage by stage, watching the numbers flow. Scroll all the way to the bottom of that machine, to where everything begins, and you will find our old friends waiting: a row of token IDs, the raw input, the grain of language from which all the apparent magic upstream is eventually built. I would not open it just yet — much of it will look like beautiful nonsense until we have built up to it together — but bookmark it. By the final chapter, every stage in that visualisation will be something you understand from the inside.

For now, sit with the central surprise of this chapter, because it is the seed of everything: a language model does not read our language at all. It reads a language it built for itself, out of the frequencies of our writing, one greedy merge at a time. Before it is a mind, or a tool, or a threat, or a marvel, it is a reader of tokens. And a token, we now know, is not a word. It is the grain into which we ground our words so that a machine could swallow them.

What did we lose in the grinding — and what, against all reasonable expectation, survived it? That question is the whole of the next chapter.

— *Neal*

↑ Back to Contents

CHAPTER 2

The Prediction Game

How Tokens Learn to Mean Something

↑ Back to Contents

At the close of the last chapter we had reduced a page of writing to a row of integers — token IDs, the grain into which language had been ground so a machine could swallow it. And I was careful to insist on something that should still be nagging at you: those integers mean *nothing*. Token 5,317 is not "closer" to token 5,318 than to token 90,001. The numbers are arbitrary labels, handed out in the order the tokenizer happened to build its vocabulary. If the model is ever going to seem to understand anything, that understanding cannot come from the numbers themselves. It has to be *built*.

This chapter is about how it gets built, and the answer comes in two movements that feel, at first, unrelated. The first is a goal — a single, almost comically modest task that the entire apparatus is trained to perform. The second is a transformation — the first thing the machine does to a token, which quietly turns a meaningless ID into something with the beginnings of meaning. By the end you will see that these two are not separate at all: the goal is what *creates* the meaning, as a kind of side effect, and that fact is the strangest and most important idea in the whole subject.

Let us take the goal first, because it is so small that its power is easy to miss.

Part I — The whole game is "what comes next?"

Here is the entire objective a large language model is trained to accomplish. Ready?

Given a sequence of tokens, predict the next one.

That is it. That is the whole game. Not "understand the passage," not "answer the question," not "reason about the world" — just: here are some tokens, guess what token comes after them. A model that has read "*the patient was prescribed a course of*" is asked, simply, to fill in the blank. Antibiotics? Steroids? Physiotherapy? It must produce its best guess, and it is scored on nothing more than whether it matched the token that actually came next in some real piece of text.

The first time you hear this, it is genuinely deflating. All of it — the essays, the code, the eerie fluency, the appearance of thought — trained on a task you would set a parrot? It sounds like a category error, as though someone claimed that memorising a phone book could teach you to hold a conversation.

And then, if you sit with it, the deflation inverts into something closer to awe. Because think, really think, about what it would take to be *good* at this game. To reliably predict the next token in *any* text drawn from the whole internet, you cannot get by on surface tricks. Consider what predicting the blank actually demands across different sentences:

```
“The capital of Kenya is ___”  →  requires a fact
“2, 4, 6, 8, ___”  →  requires a pattern
“She lied to him, so he no longer ___”  →  requires a theory of people
“def add(a, b): return a ___”  →  requires the logic of code
“The murderer, it turned out, was the ___”  →  requires holding a whole story in mind
```

To fill in the last blank correctly for a detective novel, the model would have to have tracked the plot, the clues, the misdirections — everything. The task is a keyhole, and behind the keyhole is a door that opens onto the whole of what the text was about. This is the pivotal realization of the entire field, and I want to state it as plainly as I can.

> **The back-door of prediction.** To predict the next token *well*, across all text, a system is forced to internalise whatever it takes to make that prediction — facts, grammar, patterns, cause and effect, the way people and stories behave. Understanding is not a separate goal bolted on top of prediction. It is the cheapest way to get good at prediction. We never asked the model to understand. We asked it to guess well, and understanding turned out to be the price of guessing well.

> *A machine that could truly predict the next word of an arbitrary text would, somewhere inside itself, have to understand that text — and that back-door is the entire trick.*

Everything downstream in this book — every capability, every failure — is a consequence of pushing a system relentlessly toward that one keyhole and seeing what it grows in order to fit through.

Part II — The model never gives an answer. It gives a weather forecast.

There is a subtlety here that trips up almost everyone, and getting it right now will save you endless confusion later. When we say the model "predicts the next token," it is easy to picture it confidently declaring a single winner. That is not what happens, and the difference matters enormously.

The model does not output *a* token. It outputs a *probability for every token in the vocabulary at once* — all hundred thousand of them from Chapter One. Given "*the patient was prescribed a course of*", its actual output is something like: `antibiotics` 41%, `steroids` 12%, `treatment` 9%, `physiotherapy` 4%, and a long, long tail of ever-smaller probabilities trailing off through every other token it knows, including the absurd ones. It is not an answer. It is a forecast — a full distribution of confidence spread across every possible next word.

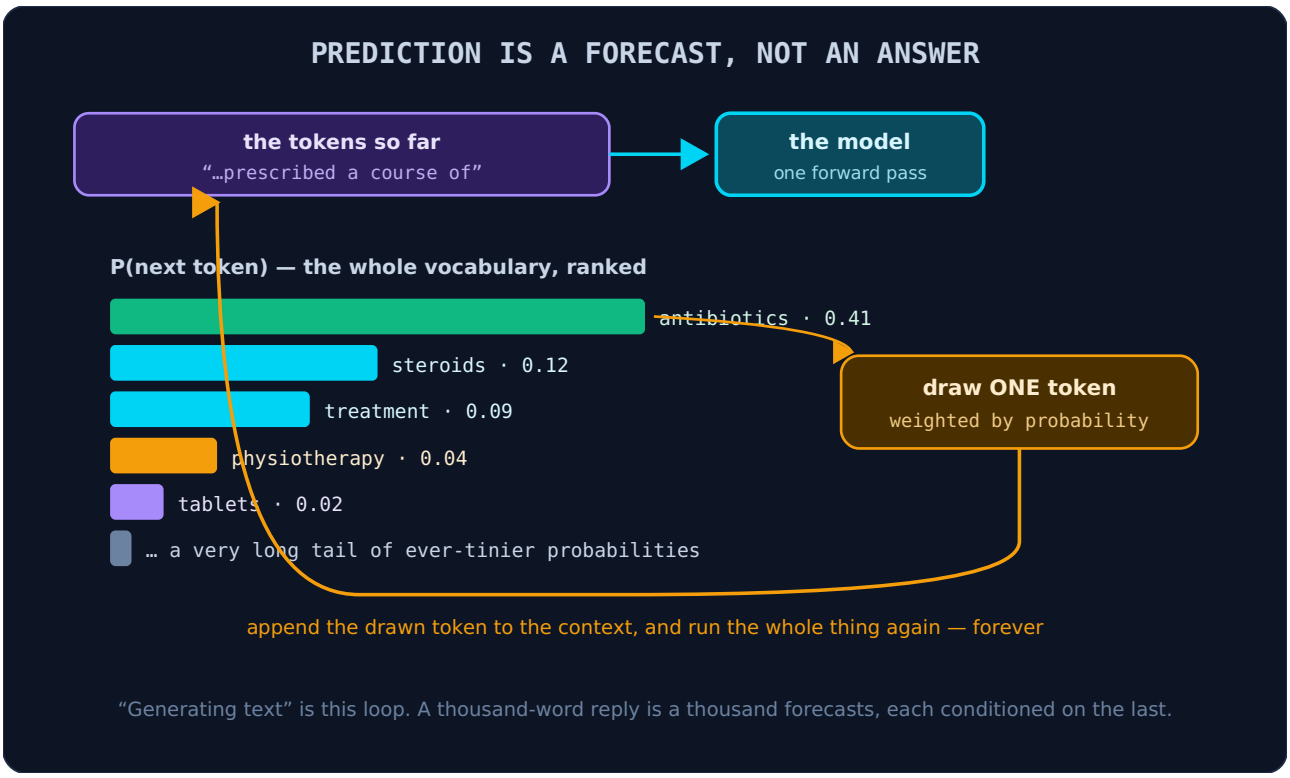

Figure 1. The generation loop. The model produces a distribution; a single token is *drawn* from it; that token is appended and the process repeats. Notice that the model does not plan a sentence — it commits to one token at a time, each choice reshaping the forecast for the next.

This forecasting nature explains a fact everyone has noticed: ask the same model the same question twice and you may get two different answers. That is not a bug, and it is not the model "changing its mind." It is because generating text involves *drawing* from the distribution, and a draw has an element of chance. How much chance is governed by a single dial you have very likely met in an API or a settings panel: **temperature**. Turn the temperature down toward zero and the model almost always takes its single most probable token — reliable, repetitive, a little robotic. Turn it up and the draw gets more adventurous, reaching further into the tail — creative, surprising, and increasingly liable to wander into nonsense. Temperature is quite literally a knob on how much you let the dice matter.

> **Why "it's just autocomplete" is both fair and deeply misleading.** The dismissive line contains a true observation — yes, mechanically, the model is completing text one probable token at a time, exactly like the suggestion strip above your phone keyboard. What the line misses is *everything about the difference in degree that becomes a difference in kind*. Your phone predicts the next word from the last two or three. This machine predicts it from thousands of prior tokens, using a model of the world it had to build in order to predict well. Calling it "just autocomplete" is like calling a surgeon "just someone with a knife." The verb is right; the noun is doing unimaginable work.

So: the goal is prediction, and the output is a forecast we sample from. But we still have not answered the question this chapter opened with. The forecast has to be *computed* from the input tokens — and the input tokens, remember, are meaningless integers. How does a meaningless integer participate in producing a rich probability distribution? For

that, we need the second movement: the transformation that gives a token its first foothold on meaning.

Part III — Giving every token a place in space

Here is the idea, and it is one of the most beautiful in all of machine learning. We are going to stop representing a token as a number, and start representing it as a *location*.

Picture a vast space — not the three dimensions we live in, but hundreds or thousands of them; hold the intuition even though you cannot picture the dimensions. Every token in the vocabulary is assigned a point in this space, specified by a list of numbers called a **vector** — its coordinates. The token `dog` is at one location; `cat` is at another; `lisinopril` at another still. This assignment — this big table mapping each of the hundred thousand tokens to its vector — is called the **embedding**, and the vectors are called embeddings too. It is, quite simply, the first thing that happens to a token once it enters the model: its ID is looked up in this table and swapped for its coordinates.

Now, why on earth would coordinates be better than a number? Because coordinates have *neighbours*, and neighbours can carry meaning. A single number line can only say "bigger" or "smaller." But a high-dimensional space can arrange its points so that **things with similar meaning sit close together**, and — this is the astonishing part — so that *directions* in the space correspond to *relationships*.

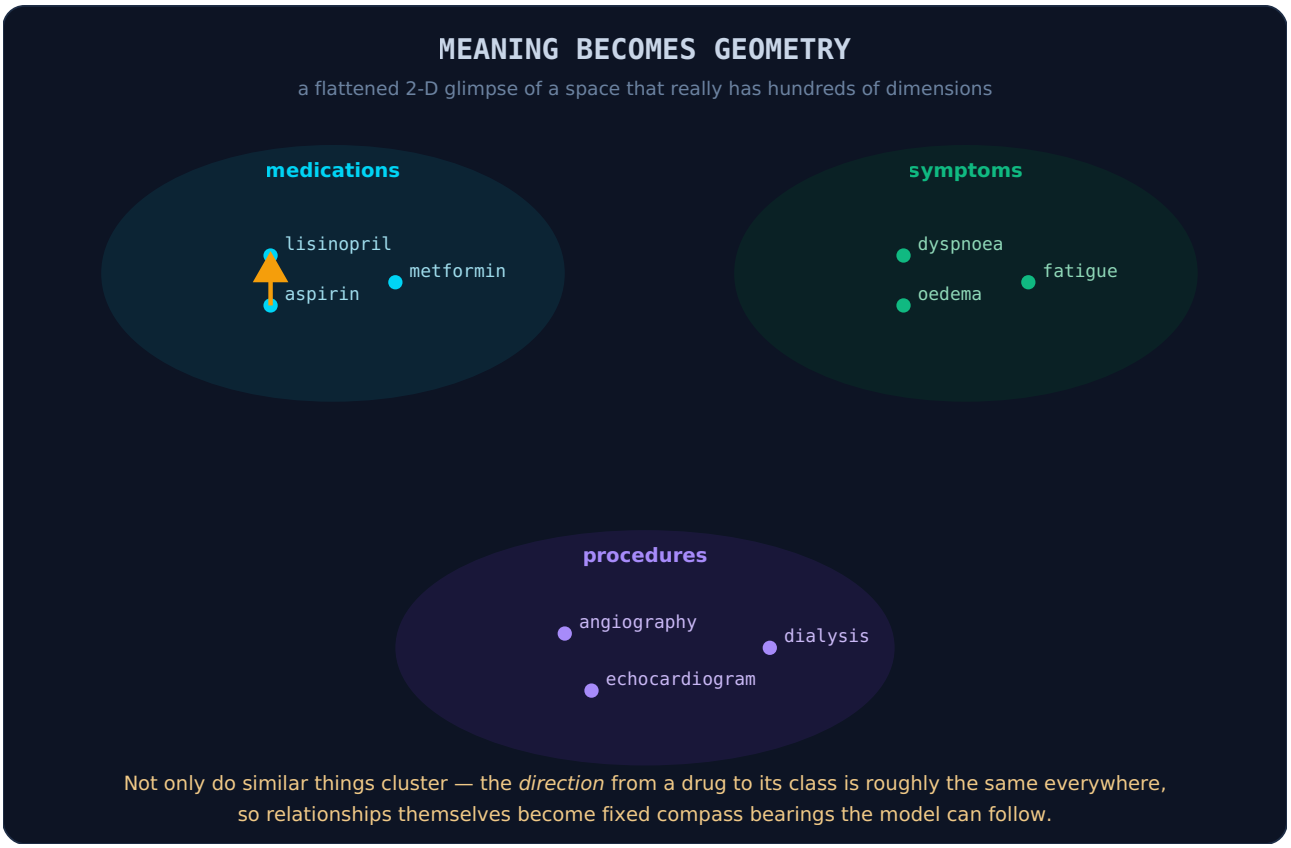

Figure 2. A cartoon of embedding space (real ones have hundreds of dimensions, which no picture can honestly show). Two properties matter: *proximity* encodes similarity, so the model can generalise from one drug to a related one it has seen less often; and *direction* encodes relationship, so the same "compass bearing" that points from a drug to its class points the same way for every drug at once.

The famous demonstration of the direction idea, in ordinary-language embeddings, is almost a magic trick: take the vector for *king*, subtract the vector for *man*, add the vector for *woman*, and you land very near the vector for *queen*. The "royalty" and the "gender" of a word turn out to be *directions* you can travel along. In a medical embedding the same structure appears in our own vocabulary: *aspirin* is to *NSAID* roughly as *lisinopril* is to *ACE-inhibitor* — the "is-a-member-of-this-drug-class" relationship is a consistent bearing in the space. The model was never handed a drug classification. It reconstructed one, as geometry, purely from how these words are used.

> **An embedding is a fixed address in a space where distance means dissimilarity of meaning.** This single move — from arbitrary ID to meaningful coordinate — is what lets a model generalise at all. Having seen "lisinopril lowers blood pressure" a thousand times, it can make a sensible guess about "ramipril" the first time, because the two live in nearly the same neighbourhood. Without embeddings, every token would be an island. With them, meaning has a landscape, and the model can navigate it.

If this idea of meaning-as-geometry feels important, that is because it is: it returns, scaled up from single tokens to whole documents, when we reach the chapter on retrieval — the mechanism behind grounding a model in your own files. The geometry you are meeting here, at the level of one token, is the same geometry that will later let a machine search a library by meaning rather than by keyword. Learn it once, here, and it pays dividends twice.

Part IV — Nobody puts the meaning there

Now for the part that genuinely borders on the uncanny, and that ties this chapter's two movements into a single knot.

Where does the arrangement of Figure 2 come from? Who decided that `aspirin` and `lisinopril` should be neighbours, that the drug-class direction should be consistent, that the space should be organised so beautifully? Surely a team of doctors sat down and placed the medical terms by hand?

No. Nobody placed anything. At the very start, before any training, the embedding table is filled with *random numbers*. Every token is flung to a random point in the space. `aspirin` and `poetry` and `dialysis` are scattered like buckshot, no cluster, no structure, no meaning whatsoever. The map begins as pure noise.

And then the prediction game begins. The model is shown ocean after ocean of real text and asked, over and over, to predict the next token — and every time it guesses, it is nudged to guess a little better. Crucially, those nudges do not only adjust some separate "prediction machinery." They reach all the way back and **move the embedding vectors themselves**. If treating `aspirin` and `ibuprofen` as neighbours helps the model predict text about pain relief, then the training gently slides their vectors together. If keeping `bank`-the-river and `bank`-the-money distinguishable helps, the training pulls on those too. Token by token, nudge by nudge, over billions of predictions, the random scatter

organises itself into the landscape of meaning — not because meaning was the goal, but because a meaningful map is what makes the prediction game winnable.

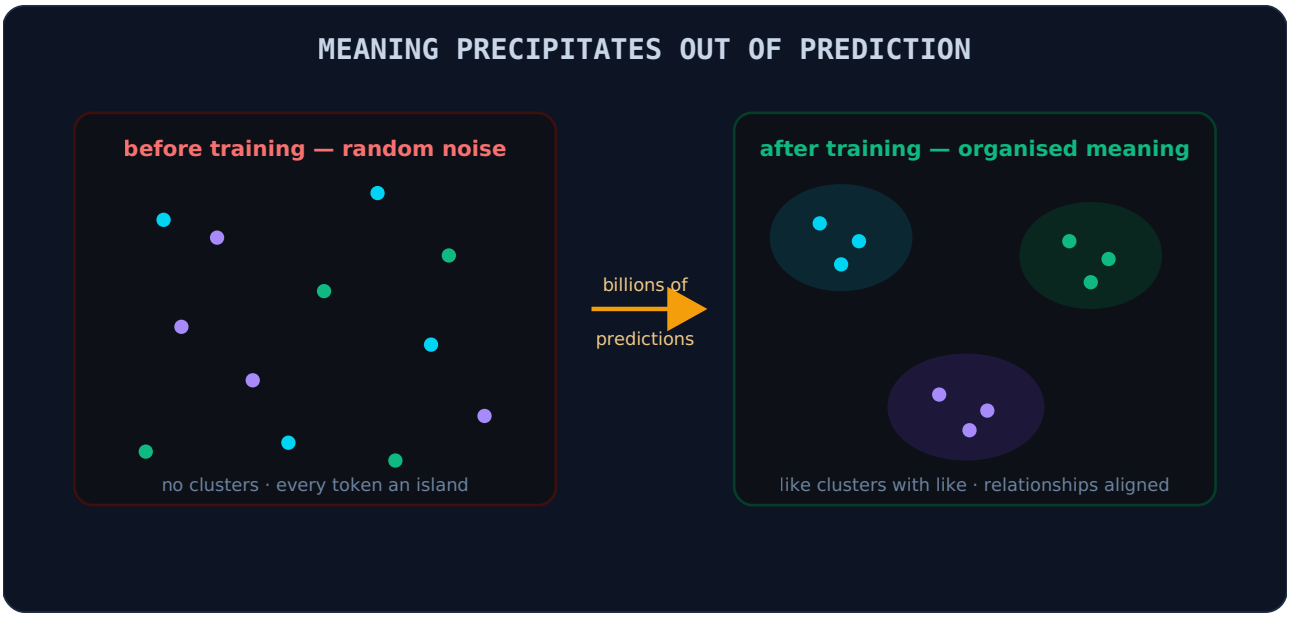

Figure 3. The map is not designed; it is grown. Training on the prediction game reaches back and rearranges the embedding vectors themselves, so that the geometry of meaning condenses out of the single pressure to guess the next token well. We will open up exactly *how* a guess reaches back and moves a vector in a later chapter on training — for now, the point is simply that meaning is discovered, not installed.

Pause on this, because it is the philosophical heart of the whole enterprise and it is almost never stated plainly. We did not teach the model what words mean. We could not have — nobody has a complete theory of what words mean. Instead we set up a game in which having a good internal model of meaning is *rewarded*, pointed the machine at a large fraction of everything humans have written, and let meaning *precipitate* out of the relentless pressure to predict. The understanding these systems have — such as it is — was not authored. It was distilled.

> **A double-edged gift for those of us in medicine.** The upside is remarkable: the model builds its own nosology, its own web of how clinical concepts relate, without ever being handed a textbook taxonomy — and that is why it can move so fluently across medicine. But look at the mechanism again. The map is distilled from *how words are used in the training text*. So if the text of the internet associates certain conditions with certain groups, certain symptoms with certain demographics, or reflects the historical blind spots of the medical literature, those associations do not merely appear in the output — they are baked into the very geometry, as distances and directions. The model's biases are not slogans it repeats. They are the shape of its space. That is a far harder thing to find, and to fix, than a bad sentence.

Part V — The gap a lone vector cannot cross

We have come a long way. Tokens have become points in a meaningful space; the space organised itself through the prediction game; and the game, we argued, is a keyhole onto genuine understanding. It would be easy to feel we are nearly done — that a model

is just a big embedding table plus some machinery to read off predictions. But there is a crack in what we have built so far, and finding it precisely is what sets up everything to come.

Here is the crack. The embedding of a token is *fixed*. `bank` gets one vector, always the same, no matter where it appears. But consider two sentences:

```
“He sat on the bank of the river and watched the current.”
“He deposited the cheque at the bank on the corner.”
```

The word `bank` means something utterly different in each — a muddy riverside in one, a financial institution in the other. Yet the embedding table, knowing nothing of context, hands over the *identical* vector for `bank` both times. A fixed point in space simply cannot be a riverbank here and a savings bank there. The single most important word for the meaning of each sentence arrives at the model wearing the same blank mask.

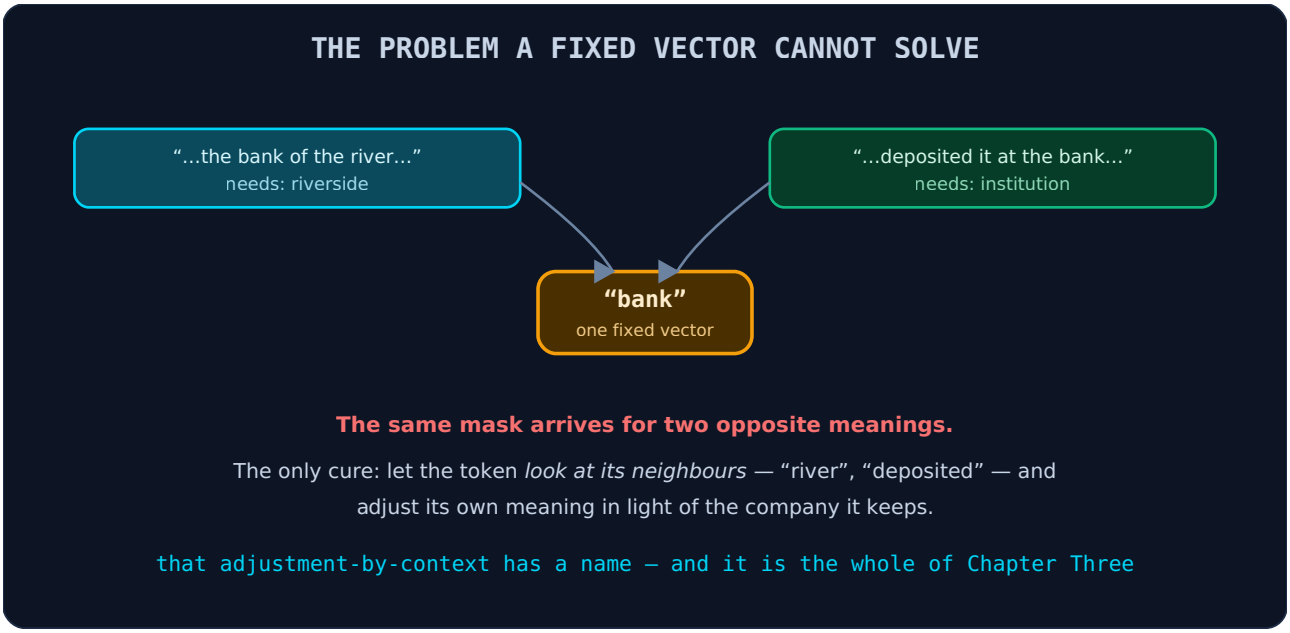

Figure 4. The limitation that motivates everything next. A static embedding captures a token's meaning *in isolation*, which is genuinely useful — but language is not made of isolated words. Meaning is contextual. To resolve "bank", the token must be allowed to gather information from the words around it and update itself accordingly. Naming that operation precisely is the doorway out of this chapter.

So we need something more. We need a token, once it has been given its starting vector, to *look around* — to see that "river" and "current" are nearby in one sentence and "deposited" and "cheque" in the other, and to let that company reshape its own meaning before any prediction is made. We need each token to stop being a fixed point and become a point that *adjusts itself in light of its neighbours*.

This is not a small tweak. It is the central mechanism of the entire architecture — the thing that separates a modern language model from every earlier attempt, the idea that made all of this work at last. It has a name, and you have heard the name even if you have never been shown what is under it.

It is called **attention**. It is the reason these models are built the way they are. And building it, from nothing, so that you understand not just what it does but *why it was the answer everyone was reaching for* — that is Chapter Three.

For now, hold the shape of what we have. A token enters as a meaningless number. It is handed a place in a vast space of meaning — a place that was not designed but grown, distilled out of the single pressure to predict what comes next. And there it sits, rich but static, wearing the same face in every sentence, waiting for permission to turn its head and look at the words around it.

We are about to grant that permission. And when we do, the whole thing comes alive.

— *Neal*

↑ Back to Contents

CHAPTER 3

Reading the Room

The Idea at the Heart of Every Language Model

↑ Back to Contents

Let me put the last chapter's unfinished business back on the table, because everything here grows out of it.

We had given every token a vector — a location in a vast space where nearness means similarity of meaning. It was a real achievement: the model could now generalise, treating `ramipril` sensibly on first sight because it lives near `lisinopril`. But the vector was *fixed*. The embedding table hands over the same coordinates for `bank` whether the sentence is about a river or a cheque. And I argued that this is fatal, because meaning in language is not a property of words in isolation. It is a property of words *in company*. The single most important token for the sense of a sentence often arrives wearing a blank mask, and the mask is identical in situations that mean opposite things.

So the requirement is clear, even before we know how to meet it. Each token must be allowed to *look at the other tokens around it* and *update its own representation in light of what it finds*. The `bank` sitting near `river` and `current` should end up shaded toward riverside; the `bank` sitting near `deposited` and `cheque` should end up shaded toward finance. Same starting vector, two different destinations, and the difference decided entirely by the company each keeps.

The mechanism that does this is called **attention**, and it is, without exaggeration, the idea that made modern language models possible. Everything before it — tokenization, embeddings — is preparation. Everything after it is elaboration. This is the hinge. So we are going to build it slowly, from the ground, and I promise that by the end it will feel not clever but *inevitable* — the obvious thing to want, arrived at by asking a sequence of reasonable questions.

Part I — The naive idea, and why it is not enough

Let us start with the simplest possible attempt, because its failure teaches us exactly what we really need.

A token must incorporate information from its neighbours. The crudest way to do that: just *average* the vectors of all the tokens in the sentence and mix a little of that average back into each token. Now every token knows, vaguely, "what kind of sentence am I in." The `bank` in the river sentence would pick up a little riverside flavour simply because `river` and `current` are in the average.

It is not a stupid idea. It is just badly undiscriminating. An average treats every neighbour as equally relevant. But when `bank` is trying to work out which sense it is, `river` is enormously relevant and the word `the` is almost totally irrelevant, and a flat

average drowns the signal in the noise. What we actually want is a *weighted* average — one where `bank` leans heavily on `river`, glances at `current`, and all but ignores `the`.

Which raises the only question that matters: **where do the weights come from?** They cannot be fixed in advance, because which words matter depends entirely on the sentence. In "the bank of the river" the relevant partner is four words away; in "he robbed a bank" it is two words away and a completely different word. The weights have to be computed *on the fly*, from the actual content of the tokens involved. A token has to be able to look out at the sentence and decide, for itself, *this one matters to me, that one does not* — and it has to make that decision based on meaning, not position.

> **The whole problem, in one sentence.** Each token needs to compute, for every other token, a *relevance weight* — how much should I let this one influence me? — where the weight depends on the *content* of both tokens, and then take a weighted average of the others accordingly. Attention is nothing more than a specific, learnable, beautifully parallel way of doing exactly this. The rest of this chapter is just filling in the "specific."

Part II — Three questions every token learns to ask

Here is the move that turns the vague wish above into a concrete mechanism, and it is genuinely elegant. To decide how much token A should attend to token B, we let each token play up to three roles, and we build each role out of the token's vector using a small learned transformation.

Think of it as each token being able to ask and answer three questions:

- A **query** — "*what am I looking for?*" This is token A's advertisement of what would be relevant to it. The `bank`-token's query might, loosely, encode "I'm an ambiguous noun; I'm looking for something that disambiguates me — a body of water, or money."
- A **key** — "*what do I offer?*" This is each token's advertisement of what it contains, phrased so that a matching query will recognise it. The `river`-token's key encodes "I am about water, geography, nature."
- A **value** — "*what will I actually hand over if you attend to me?*" This is the information a token contributes once it has been selected — the payload.

Each of these three is produced from the token's embedding by its own learned projection — three different "views" of the same underlying vector, each shaped during training to be good at its job.

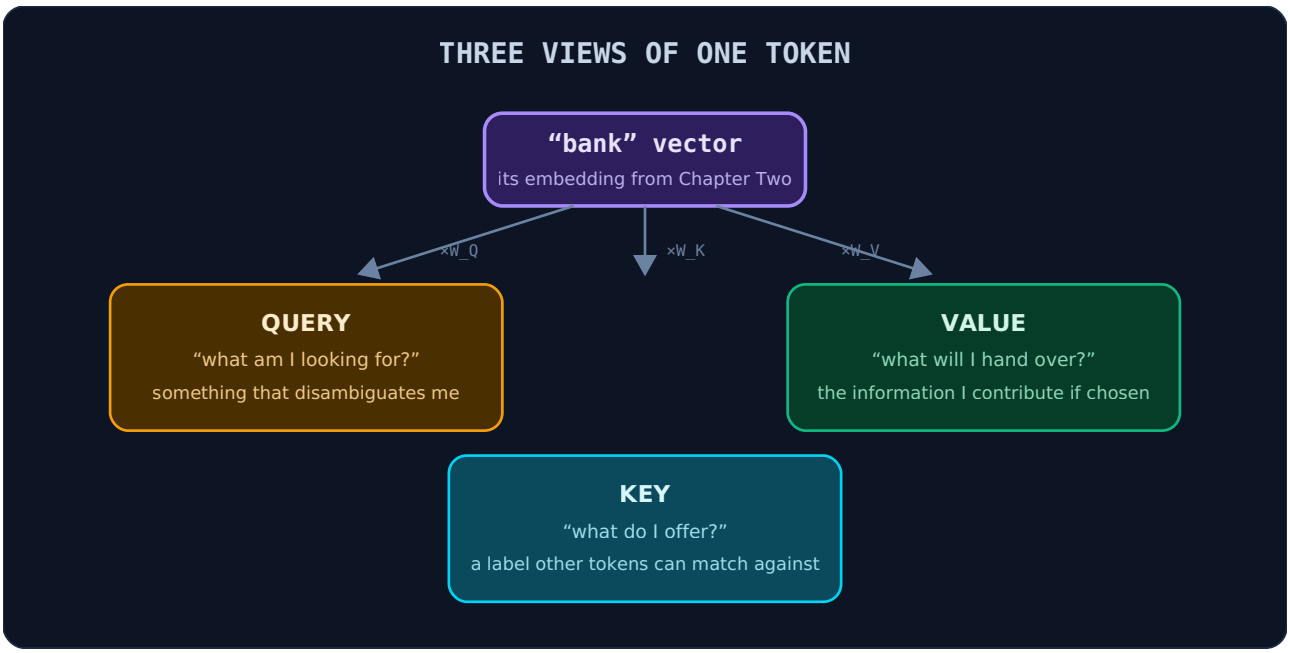

Figure 1. Every token is projected into three roles by three learned matrices (W_Q, W_K, W_V). The same word plays all three parts at once: it asks its own question (query), advertises itself to others (key), and stands ready to contribute (value). These three projections are among the parameters the model learns during training — the model discovers, from data, what makes a good question and a good answer.

The trick, now, is how a query and a key are compared. Both are vectors — arrows in the same space — and there is a standard way to measure how well two vectors align: the **dot product**, which is large and positive when two vectors point the same way, near zero when they are unrelated, and negative when they point apart. So the relevance of token B to token A is simply: take A's query, take B's key, and dot them together. A high number means "B is exactly the kind of thing A was looking for." That single number is the raw ingredient of every attention weight.

Part III — From raw scores to a spotlight

Let us make this concrete with the sentence that started us off. The `bank`-token issues its query, and we score it against the key of every token in the sentence — including itself. We get a row of raw numbers, one per token: high for `river`, moderate for `current`, low for `the` and `of`.

Raw scores, though, are awkward to use directly. They can be any size, positive or negative, and they do not add up to anything meaningful. What we want is to turn them into a clean set of *weights* — all positive, and summing to exactly one — so they behave like a spotlight of fixed total brightness that we can shine across the sentence, pouring most of it on the relevant words and only a sliver on the rest. The function that does this conversion is called the **softmax**, and while its formula is a small piece of arithmetic, all you need to hold is what it accomplishes: it takes any row of scores and squashes it into a tidy distribution of attention that sums to one, exaggerating the gaps so the strong scores get the lion's share.

Figure 2. The scoring step for a single token. Its query is dotted against every key to produce raw scores; softmax turns those into attention weights that sum to one. Crucially, these weights are not stored anywhere — they are recomputed from scratch, from the actual content of this particular sentence, every single time. Change one word and the whole spotlight redraws.

We now have, for our `bank`-token, a spotlight: 62% on `river`, 21% on `current`, and dribs and drabs elsewhere. The final step is the one we were aiming for all along. We take a *weighted average of the value vectors* — 62% of `river`'s value, 21% of `current`'s value, and so on — and that blended vector becomes the information `bank` has gathered from its context. Mixed back into `bank`'s own representation, it drags the meaning firmly toward riverside. The blank mask has been painted.

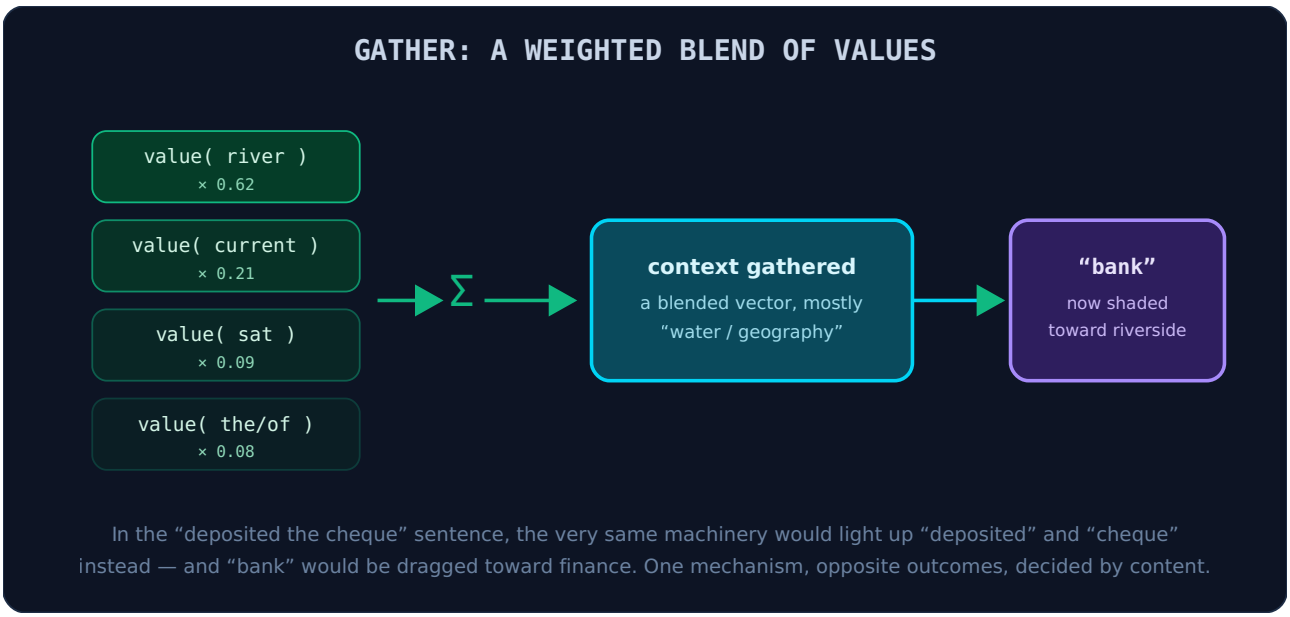

Figure 3. The payoff. The token's updated representation is a weighted sum of the *value* vectors of everything it attended to. The static embedding of Chapter Two has become *contextual*: “bank” is no longer a fixed point but a point that has moved, in this sentence, toward the meaning its neighbours implied. The crack we ended the last chapter on is healed.

And that — query, key, value; score, softmax, blend — is the entire mechanism. Read those two figures again and notice there is no magic anywhere in them, only a learnable, content-driven weighted average. Every token in the sentence does this simultaneously, each issuing its own query, each ending up with its own context-updated representation. The sentence walks in as a row of frozen, isolated vectors and walks out as a row of vectors that have *talked to each other*.

> **The ward-round analogy, because it genuinely fits.** Picture a multidisciplinary team meeting. Each patient's case (a token) silently broadcasts a question — *“what here is relevant to me?”* (its query). Every other case advertises what it holds — *“I'm the one with the deranged clotting”* (its key). Each case then listens, weights the others by relevance, and updates its own plan with a blend of what the relevant cases contribute (their values). Nobody reads every chart end to end; relevance is negotiated by content, in parallel, all at once. And — the part worth remembering — when a model attends to the *wrong* case, confidently attributing a fact to the wrong antecedent, you are watching an attention weight land in the wrong place. A surprising share of an LLM's plausible-but-wrong answers are, at bottom, attention pointed at the wrong word.

Part IV — Why this was the breakthrough

It is worth pausing to appreciate *why* this particular design changed everything, because it did not merely work — it unlocked the entire scale of the modern era. Two properties are responsible.

The first is that attention is **content-addressed**, not position-addressed. Older approaches to sequence modelling processed text strictly left to right, carrying a running summary forward and asking, in effect, "what was a few steps back?" That

makes reaching across long distances hard — by the time you are five hundred words in, the details of word twelve have been squeezed to mush. Attention asks a completely different question: not "who is near me?" but "who *matches* me?" A token can reach directly across a thousand others in a single step to the one word that resolves it, with no decay over distance. The same machinery, unchanged, handles a pronoun and its antecedent, a subject and its far-off verb, a closing bracket and its opening one, and a fact stated three paragraphs earlier. It never needed to be told which of these tasks it was doing; relevance is relevance.

The second is that attention is **completely parallel**. Because every token computes its query, its key, and its value independently, and because all the scores are just one big multiplication of queries against keys, the entire operation runs at once — not word by word, but the whole sentence in a single sweep. This is not a minor efficiency note. It is the reason these models could be trained on a meaningful fraction of the internet at all. The architecture finally matched what modern hardware is savagely good at: doing enormous numbers of multiplications simultaneously. An idea that was merely elegant would have stayed in a research paper. An idea that was elegant *and* parallel became the foundation of an industry.

Part V — The one rule: you may not read the future

There is a single, crucial constraint I have been quietly deferring, and now it clicks into place with everything from Chapter Two.

Recall the game: the model is trained to predict the *next* token from the ones before it. Now imagine we let a token attend freely to every other token in the sentence, including the ones that come after it. When the model is trying to predict the word after `bank`, it could simply *peek* at that word through attention — and the whole exercise collapses into cheating. It would ace training by copying the answer and learn nothing.

So we impose one rule. When a token computes its attention, it is permitted to look only at itself and the tokens *before* it — never at the ones that follow. Before the softmax, the scores for all future positions are blanked out entirely, so they receive exactly zero weight. Information is allowed to flow backward-to-forward, never forward-to-backward. This is called the **causal mask**, and it is why this whole family of models earns the name *autoregressive*: each position is built only from the past, so that predicting the future remains an honest test.

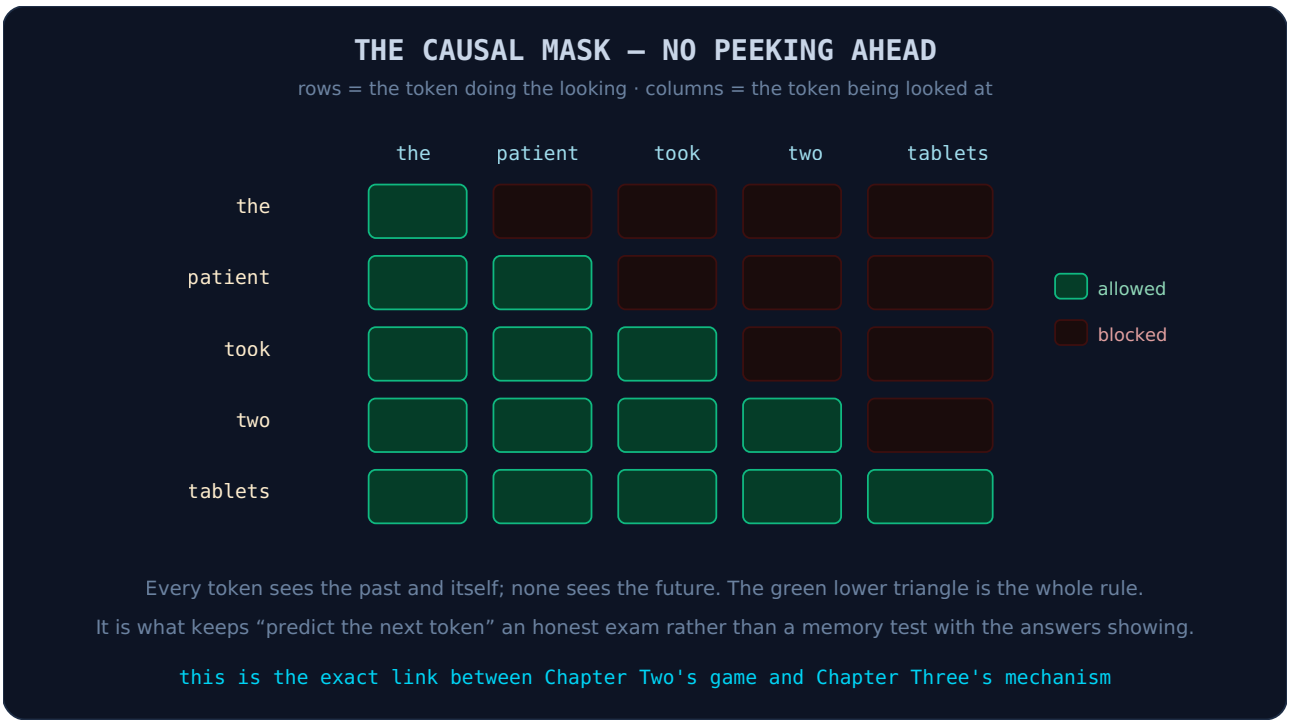

Figure 4. The causal mask, drawn as a grid of who-may-look-at-whom. The permitted cells form a lower triangle: each token attends to itself and everything before it, never after. This single constraint is the seam that stitches attention to the prediction objective — it is why a model can be trained to predict the future without ever being shown it.

Part VI — What one glance cannot do

We have built something remarkable, and it is worth being precise about exactly how far it gets us — because the honest limits are what point straight at the next chapter.

Our attention mechanism lets every token look once across the sentence, gather the relevant context, and update itself. The frozen vectors of Chapter Two are now fluid, context-aware, alive to their surroundings. But look closely at two things it does *not* yet do.

First, a single attention operation can only track *one kind of relationship at a time*. Its one set of query and key projections learns one notion of "relevant" — perhaps "which earlier noun does this pronoun refer to." But a sentence is threaded with many relationships at once: grammatical agreement, who-did-what-to-whom, the matching of quotation marks, the long-range echo of a topic. One query-key scheme cannot be all of these simultaneously. We are going to need *many* attention operations running in parallel, each free to specialise in a different kind of relationship — and then a way to combine what they each found. That plurality has a name, *multi-head attention*, and it is the first thing we build next.

Second — and this is subtler — attention *moves information around* but does very little *thinking* with it. It is a superb librarian, fetching the right passages and laying them on the desk, but fetching is not the same as reasoning over what was fetched. After each token has gathered its context, it needs a moment to sit and *compute* on what it now holds — to transform the gathered information into something new. That step, a small

computation applied to each token on its own, alternates with attention in a rhythm that repeats: *gather, then think; gather, then think*. Stack that rhythm dozens of times and you have the full engine — the **transformer block**, and the deep tower built from it.

So this is where we stand. We have the beating heart. We have watched a word turn its head, read the room, and change its meaning. What we do not yet have is the whole body: the many parallel glances, the per-token thinking that turns gathered context into inference, and the depth — the stacking of block upon block — that lets meaning be refined layer after layer until a confident next-token forecast can finally be read off the top.

Building that body, from this heart, is Chapter Four. And when it is finished, you will be able to look at a diagram of a real transformer — the kind that has intimidated readers for years — and find, in every box, something you understand from the inside.

— *Neal*

↑ Back to Contents

CHAPTER 4

The Tower

How a Transformer Turns Attention into Thought

↑ Back to Contents

At the end of the last chapter we had built something genuinely powerful and left it deliberately incomplete. We had attention: the mechanism by which a token issues a query, matches it against the keys of every earlier token, and updates itself with a weighted blend of their values. A word could finally read the room. The frozen vectors of Chapter Two had come alive to their context.

But I was careful to name two things attention *cannot* do on its own, because they are the exact shape of what we build now. First, a single attention operation tracks only one kind of relationship at a time — one notion of "relevant" — while real language is threaded with many relationships at once. Second, and more subtly, attention *moves information around* but does very little *thinking* with it; it is a superb librarian fetching the right passages onto the desk, but fetching is not the same as reasoning over what was fetched.

This chapter fixes both, and then does something more ambitious than either fix: it discovers that once you have a unit which gathers context and then thinks about it, the natural thing to do is *stack that unit, deep* — and that depth is where a transformer's apparent intelligence actually lives. We are going to build the block, and then build the tower.

There is also a small debt to settle before we begin — a gap I let you walk past in the last two chapters, and which you may already have felt nagging. Let us pay it first, because it is quick and it matters.

Part I — A debt: the machine does not know word order

Look back at the attention mechanism of Chapter Three and ask a sharp question: *does it know which word came first?*

Astonishingly, no. Attention compares every query to every key by content alone. Nothing in "query dotted against key" encodes *where* in the sentence each token sits. If you shuffled the words of a sentence, the set of queries, keys, and values would be exactly the same set — only reordered — and the attention machinery, which is blind to order, would compute essentially the same blend. But "the dog bit the man" and "the man bit the dog" contain identical words and mean opposite things. A mechanism that cannot tell them apart cannot possibly model language.

The fix is disarmingly direct. Since the token vectors carry no sense of position, we simply *add* position information into them before they ever enter the first attention layer. Alongside the embedding table from Chapter Two — which answers "what does this

token mean?" — the model keeps a second source that answers "where does this token sit?", and the two are added together so that each token's starting vector encodes both its identity and its place in line. Position stops being invisible; it becomes part of what every query and key is built from, so a token can now attend not only by meaning but by *where things are* relative to it.

> **Two facts, one vector.** Before the first block, each token's vector is the sum of two things: *what it is* (its meaning-embedding) and *where it is* (its position). From that point on, everything — every query, key, and value in every layer — is computed from vectors that already carry both. Order is no longer lost; it is baked in at the door. Debt paid. Now we can build.

Part II — Many glances at once

Return to the first limitation: one attention operation can only follow one thread of relevance. In the sentence "*the surgeon told the registrar that she had made an error,*" several questions hang in the air simultaneously. Who does "she" refer to — the surgeon or the registrar? Which verb governs "error"? What is the emotional and hierarchical relationship between the two people? A single query-key scheme has to commit to *one* notion of relevance, and cannot chase all of these at once.

So we do the obvious, generous thing: we run *many* attention operations in parallel, side by side, each with its own independent query, key, and value projections. Each one is called a **head**. Because their projections are learned separately, the heads are free to specialise — and, remarkably, when you inspect trained models they demonstrably do. Interpretability researchers, dissecting real networks, reliably find heads that have quietly taken up distinct jobs: one tracks which noun a pronoun refers to, another matches opening brackets to closing ones in code, another attends to the previous occurrence of the current word, another follows subject-verb agreement across long distances. Nobody assigned these roles. The heads discovered a division of labour, the way an experienced team settles, without discussion, into who watches what.

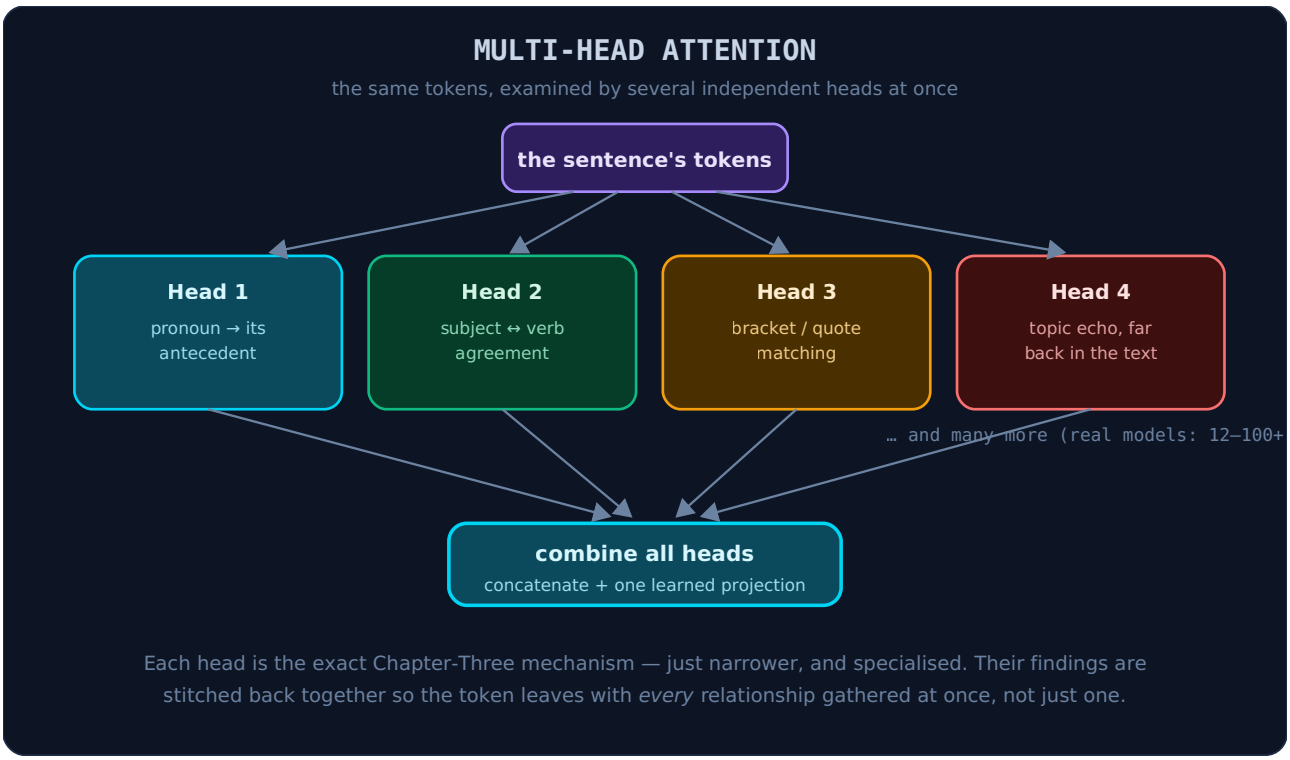

Figure 1. Multi-head attention. Rather than one glance, the model takes many simultaneously, each head a full copy of the Chapter-Three mechanism with its own projections and its own learned speciality. Their outputs are concatenated and passed through a single combining projection, so each token emerges having consulted the sentence along many independent lines of relevance at the same time.

That is the first limitation dissolved. A token no longer follows a single thread; it follows a whole braid of them, in parallel, and leaves with all of them woven into its updated representation.

Part III — The thinking step

Now the subtler limitation, and the part of the transformer that most explanations skate over even though it holds the majority of the model's substance.

Attention, even multi-headed, is fundamentally an act of *gathering*. It reaches out, collects the relevant values from around the sentence, and blends them in. But blending is not reasoning. Having assembled "this token is a pronoun, its antecedent is the surgeon, the mood is one of admitted fault," something must now *do something* with that assembled picture — transform it, draw an inference from it, turn the gathered ingredients into a new conclusion. Gathering sets the table; someone still has to cook.

That cooking is done by the second half of the block: a small, ordinary neural network applied to **each token independently**, once its context has been gathered. It is often called the feed-forward layer or the MLP, and mechanically it is the simplest thing in the whole architecture — take the token's vector, pass it through one expanding transformation and a nonlinearity and one contracting transformation, and out comes a new vector. No looking at other tokens; no attention. Each token, alone, thinks about what it has just gathered.

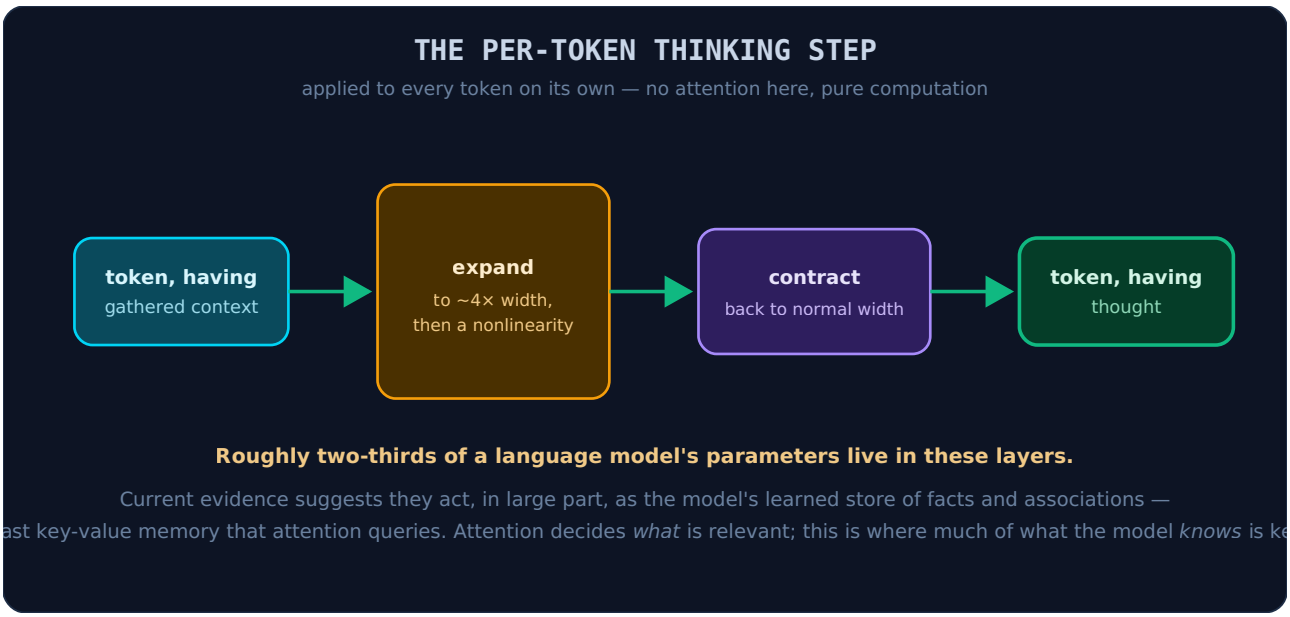

Figure 2. The feed-forward step — the block's "compute" half. It is applied identically and independently to every token. Mechanically trivial, it is nonetheless where the majority of the model's parameters reside, and there is good evidence it functions substantially as an associative memory: the place where facts learned in training are stored and retrieved. Attention gathers; this thinks.

So the block has a rhythm, and it is worth naming because it is the whole design in four words: **communicate, then compute.** First the tokens talk to each other and gather what they need (multi-head attention); then each retreats and thinks about what it gathered (the feed-forward network). Reach out; reflect. Gather; digest. It is a rhythm any clinician will recognise from the ward round itself — first you collect the relevant information from everyone around the bed, then you step back and reason about the case on your own.

> **Where the knowledge actually sits.** It surprises people that attention — the celebrated part — holds relatively few of a model's parameters, while the humble feed-forward layers hold most of them. But it fits a clinical intuition. Your ability to *gather* the right facts about a patient is a skill; your *store* of medical knowledge is a vast, slowly-built library. The transformer separates these too: attention is the gathering skill, the feed-forward layers are the library. When a model "knows" that a particular drug prolongs the QT interval, that fact most likely lives, encoded, in these layers — retrieved when attention routes the right query to it.

Part IV — The stream that holds it all together

We now have the two working halves. But how are they wired together — and stacked, dozens of times, without the signal degrading into noise? The answer is one more idea, quiet but load-bearing, and it changes how you should picture the whole machine.

Do not picture the block as a pipe that transforms a token and passes on a *replacement*. Picture instead a **residual stream**: a running vector for each token that flows straight up through the entire model like a central highway, and onto which each sub-layer *adds* its contribution rather than overwriting what was there. Attention does not replace the token; it reads the current state of the stream and *adds* what it gathered back onto it.

The feed-forward layer does not replace the token; it reads the stream and *adds* its computed refinement. The stream is never wiped and rewritten — it is a chart that each stage annotates and passes upward, thicker with meaning at every step.

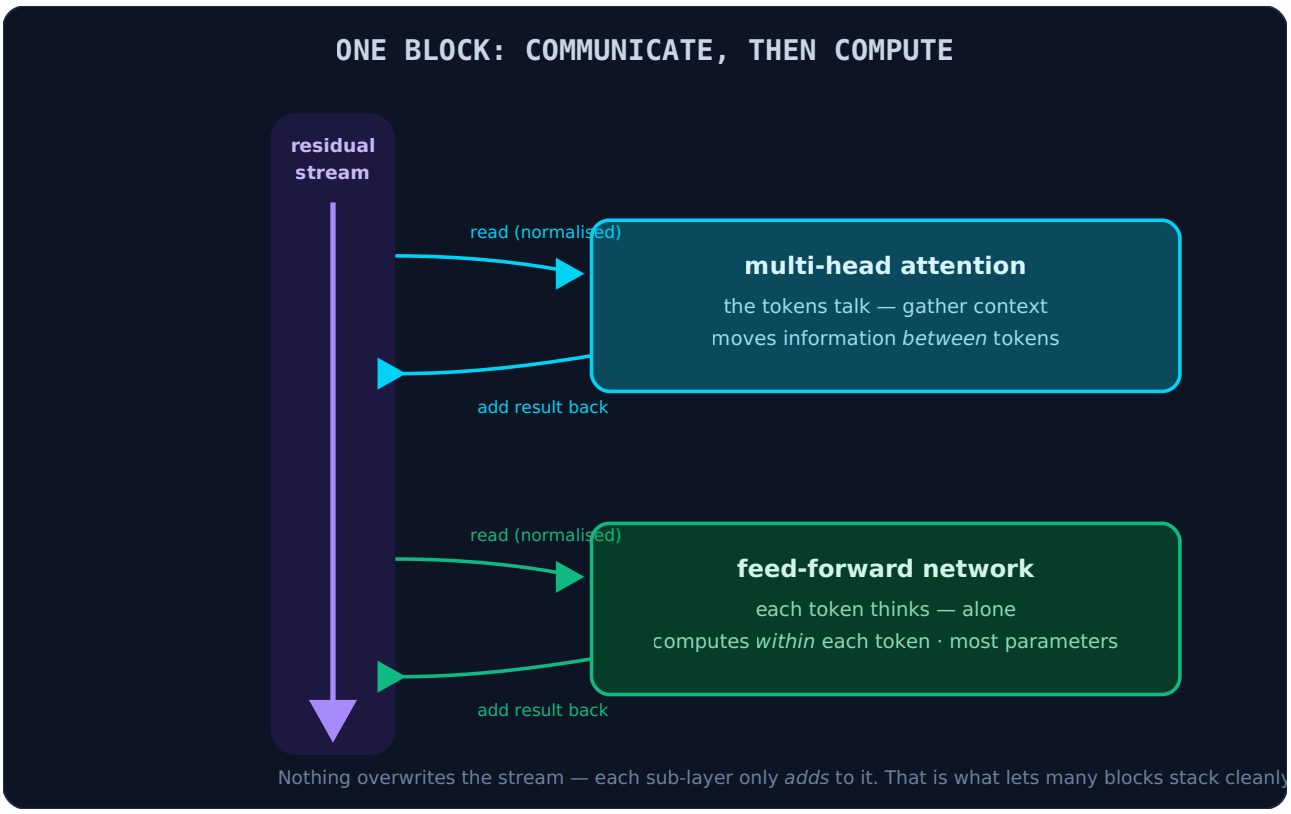

Figure 3. The complete transformer block. The residual stream runs straight up the middle; attention and the feed-forward network each read a normalised copy of it and add their contribution back. "Normalised" refers to layer normalization — a routine rescaling of each token's vector to a stable spread before it is used, the statistical hygiene that keeps very deep stacks trainable. The important shape is the addition: contributions accumulate, they do not replace.

Why addition, and why does it matter so much? Two reasons, one practical and one conceptual. Practically, adding rather than overwriting gives the training signal a clean, unobstructed path all the way down through even a hundred stacked blocks — the reason very deep networks became trainable at all, and a point we will feel the force of in the next chapter on training. Conceptually, the residual stream lets each block make a *modest edit* to a shared, evolving representation rather than reinventing it from scratch. A block does not have to solve the whole sentence; it just has to add one useful annotation and pass the chart along. Small edits, stacked deep, compound into something that looks remarkably like understanding.

Part V — Depth: the tower

And now the step that turns a clever unit into a mind-like thing. We take the block — attention, then feed-forward, both adding to the residual stream — and we **stack it**. The output stream of one block is the input stream of the next, unchanged in shape, so the

blocks pile up cleanly: a dozen high in a small model, dozens in a large one, well past a hundred at the frontier. This is the *tower*, and depth is where the magic quietly happens.

Watch what accumulates as a token's vector rises through the stack. In the earliest blocks, the annotations are close to the surface — resolving a word's part of speech, binding a pronoun, noting simple local structure. A little higher, the edits grow more abstract — assembling phrases into clauses, tracking who did what to whom, registering tone. Higher still, the representation carries things no single word contains — the gist of an argument, the state of a story, the constraint that the next word must satisfy.
Meaning is not computed in one heroic step. It is *refined*, layer upon layer, each block adding a slightly more abstract reading of the same underlying text, until near the top the stream holds a representation rich enough that the single most useful thing left to do is predict what comes next.

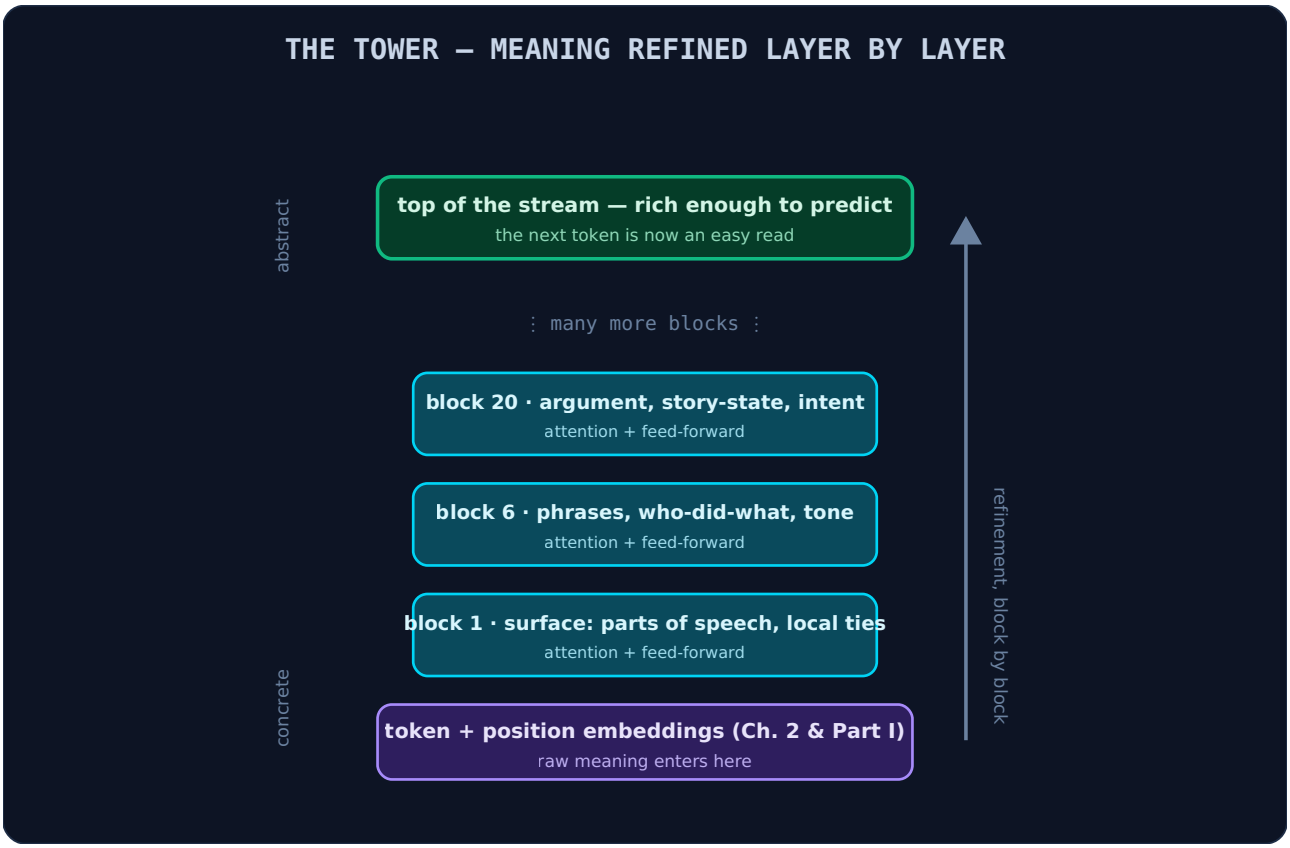

Figure 4. Depth as refinement. The same token vector rises through the stack, each block adding a slightly more abstract reading. The labels here are a useful cartoon, not a precise map — the division of labour across real layers is blurrier and is still being charted by interpretability research — but the direction of travel is real: concrete near the bottom, abstract near the top. Depth is not repetition; it is elaboration.

Depth is the quiet source of the apparent intelligence. A single block is not smart. What makes a transformer capable is dozens of them in sequence, each permitted to gather afresh and think afresh on a representation the layers below have already enriched. "Reasoning," to whatever degree these models have it, is not a special module bolted on — it is what happens when you refine a representation deeply enough. This is also why bigger models, with more layers and width, tend to be more capable: more depth is more room to refine.

Part VI — Reading the answer, and closing the loop

We have built the tower. One final, almost anticlimactic step converts it back into the thing Chapter Two promised: a forecast.

At the very top of the stack, we take the residual-stream vector — but only at the *last* position, the token after which we want to predict — and we project it, through one more learned transformation, out to a list of scores with one entry for *every token in the vocabulary* from Chapter One. Softmax turns that list of scores into a probability distribution, and there it is: `antibiotics` 41%, `steroids` 12%, the whole forecast we drew in Chapter Two, now actually *computed*, from the bottom of the tower to the top. The machine has read the tokens, refined their meaning through dozens of layers of communicate-then-compute, and read off, at the summit, its best guess at what comes next.

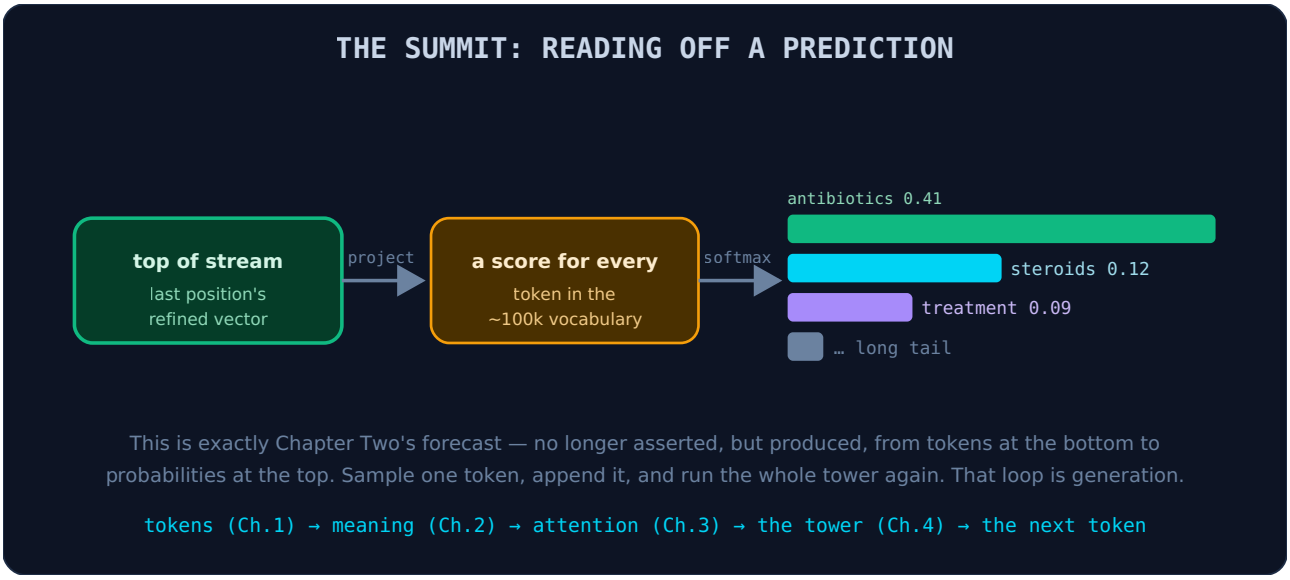

Figure 5. The loop closes. The refined vector at the top of the tower is projected to one score per vocabulary token and softmaxed into the next-token distribution — the very forecast Chapter Two described. Sample from it, append the chosen token, and the entire tower runs again for the next word. Four chapters, and the full path from raw text to generated text now stands complete.

Take a moment with what we have assembled, because it is the whole thing. Text becomes tokens (Chapter One). Tokens become vectors in a space of meaning, carrying their positions (Chapter Two, and Part I here). Those vectors flow up a deep tower, and at every level they gather context from one another through many-headed attention (Chapter Three) and then think privately in the feed-forward layers (this chapter), adding refinement after refinement to a residual stream that is never overwritten, only enriched. At the summit, the enriched vector is read off as a forecast of the next token. That is a large language model. There is no other secret ingredient. Everything else — the scale, the training, the polish — is in service of this architecture, not in addition to it.

> **Now go and see it whole.** In Chapter One I asked you to bookmark an interactive 3-D visualiser of a working language model and warned that most of it would look like beautiful nonsense until we had built up to it. Open it again now. The embeddings at the bottom, the attention heads with their query-key-value projections, the feed-forward layers, the residual stream running up the middle, the stack of blocks, the final projection to logits at the top — every stage in that visualisation is now something you can name and explain. That is not a small thing. Most people who use these tools daily could not label a single box. You can label all of them.

Part VII — The machine is built, but it is empty

And yet — for all that we have assembled — we have not actually explained the most important thing.

Go back and count the pieces that were simply *given* to us, fully formed, throughout these four chapters. The embedding table that places `aspirin` near `ibuprofen`. The query, key, and value projections that let attention know what is relevant. The feed-forward layers that hold the model's knowledge. The combining projections, the output projection, every last one of the millions or billions of numbers that make this tower do anything at all. Where did they come from? Who set them?

We already glimpsed the unsettling answer back in Chapter Two: *nobody set them.* They all begin as random noise. The magnificent engine we have just built, if you constructed it today and ran a sentence through it, would produce pure gibberish — a confident forecast of an entirely random next token — because every projection in every head, every weight in every feed-forward layer, is meaningless static. We have built the body in exquisite detail. It has no mind in it yet.

What fills it — what turns billions of random numbers into an embedding space that knows medicine and attention heads that track pronouns and feed-forward layers that store facts — is *training*: the relentless process by which the single pressure to predict the next token, applied across a large fraction of everything humans have written, reaches back through this entire tower and nudges every one of those numbers, billions of times, until gibberish becomes fluency. We have deferred the mechanism of that "reaching back" three times now. We can defer it no longer.

How a guess at a single next token can reach all the way down through a hundred layers and correct every parameter that contributed to it — how noise becomes knowledge — is the subject, at last, of Chapter Five.

— Neal

↑ Back to Contents

CHAPTER 5

How Noise Becomes Knowledge

Training a Language Model

↑ Back to Contents

Let me begin by making the problem vivid, because its difficulty is easy to underestimate and the solution is easy to take for granted once you have seen it.

At the end of the last chapter we had assembled a complete transformer — the embedding table, the attention heads with their query, key, and value projections, the feed-forward layers, the residual stream, the whole tower, and the final projection that reads a prediction off the top. It is an engine of real intricacy. And I told you the uncomfortable truth: if you built it today and ran a sentence through it, it would produce *gibberish*. Every one of its parameters — and there may be billions of them — begins life as a random number. The embedding that we later admired for placing `aspirin` near `ibuprofen` starts as buckshot. The attention heads that we said track pronouns start attending to nothing in particular. The feed-forward layers that hold the model's knowledge start holding noise.

So here is the question this chapter answers, and I want you to feel how audacious it is. We have a machine with a billion knobs, all set randomly. We want to turn those random settings into precisely the configuration that makes the machine fluent, knowledgeable, and startlingly capable. Nobody can set a billion knobs by hand, and nobody knows what the right settings even are. How, then, does the machine find them *itself*?

The answer is one of the most beautiful procedures humans have ever devised, and it has just three moving parts: a way to *measure* how wrong the machine currently is, a way to work out which direction to nudge every knob to make it *less* wrong, and the patience to do this a truly staggering number of times. Measure, nudge, repeat. That is all training is. Let us build each part.

Part I — Measuring wrongness

You cannot improve what you cannot measure, so everything starts with turning "the model is wrong" into a single, honest number.

We have exactly what we need, and we have had it since Chapter One. Recall that language modelling is *self-supervised*: we take real text, and at every position the "correct answer" is simply the token that actually came next. So we can play the following game against any piece of writing. Show the model the text up to some point. Let it produce its forecast — the full probability distribution over the vocabulary from Chapter Two. Then look at the token that *truly* came next, and ask a pointed question: **how much probability did the model put on the truth?**

If the model was confident and right — it put 70% of its forecast on the token that actually followed — it was barely wrong at all. If it was confident and *wrong* — it put 70% on some other token and a measly 0.1% on the truth — it was badly, embarrassingly wrong. The measure of wrongness we use, called the **cross-entropy loss**, captures exactly this: it is large when the model assigned low probability to the true next token, and small when it assigned high probability. You can think of it, without losing anything important, as a number that measures the model's *surprise* at the correct answer. A good model is rarely surprised by what comes next; a bad one is constantly astonished.

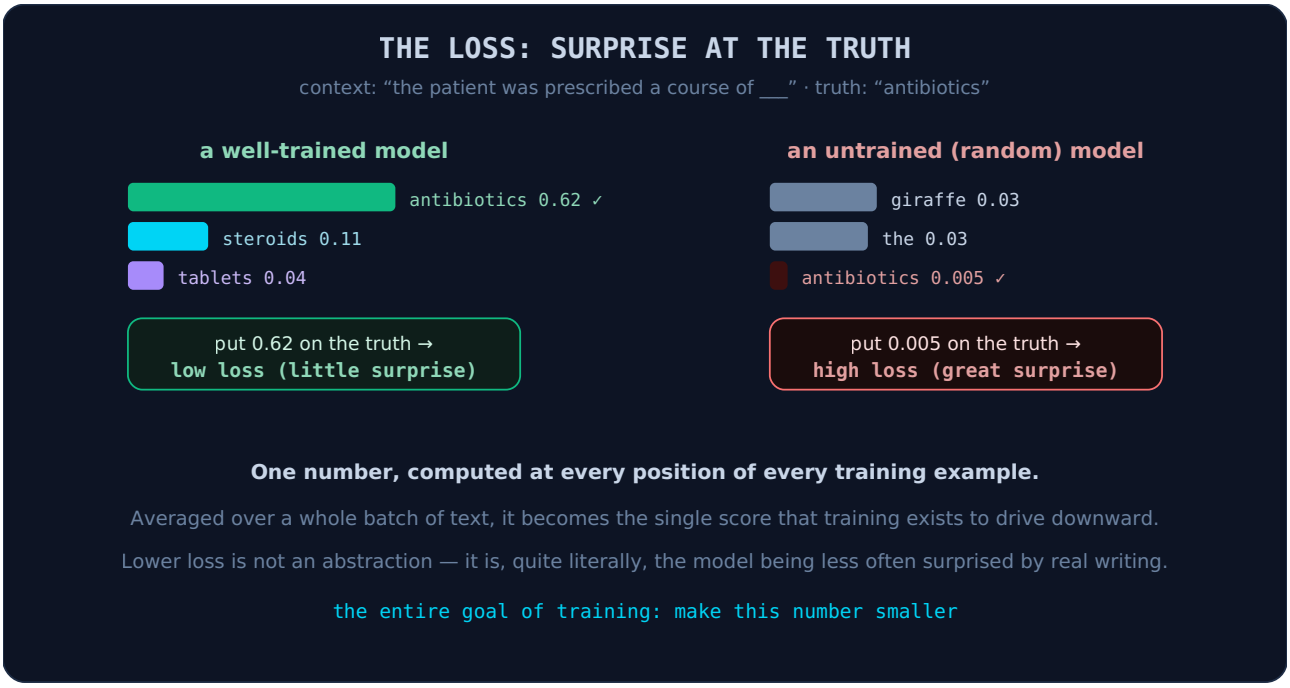

Figure 1. The loss turns "how wrong is the model right now?" into one number, by asking how much probability it placed on the token that actually came next. It is high for a confidently-wrong model and low for a confidently-right one. This single number — averaged over many predictions — is the compass for everything that follows.

So now we have a number. For any setting of the billion knobs, we can compute a loss — a measure of how badly, on average, the machine currently predicts real text. And the entire problem of training has been transformed into something almost tractable-sounding: *find the setting of the knobs that makes this number as small as possible.* We have turned "teach a machine to understand language" into "minimise a number." That reframing is the whole ballgame.

Part II — Which way is downhill?

Here is a way to picture the task that will carry you a long way. Imagine the loss as a *landscape*. Every possible setting of the billion knobs is a location, and the height of the terrain at that location is the loss — how wrong the model is with those settings. A random starting point is somewhere high up on a hillside, in the fog. The configuration we want — low loss, fluent model — is down in a valley we cannot see. Training is the

search for that valley. And the rule for the search is the simplest imaginable: *always step downhill.*

But downhill in a landscape of a billion dimensions? You cannot see it, cannot picture it, cannot survey it. What you *can* do, remarkably, is feel the slope directly under your feet. For each of the billion knobs, calculus lets us ask a precise local question: *if I nudged this one knob a hair in the positive direction, would the loss go up or down, and how steeply?* Collect that answer for every knob at once and you have the **gradient** — a vector that points in the direction of steepest *increase* of the loss. And if the gradient points most steeply uphill, then its exact opposite points most steeply downhill. So we take a small step in that opposite direction, and we have, with certainty, reduced the loss a little. Every knob moves a touch toward "less wrong."

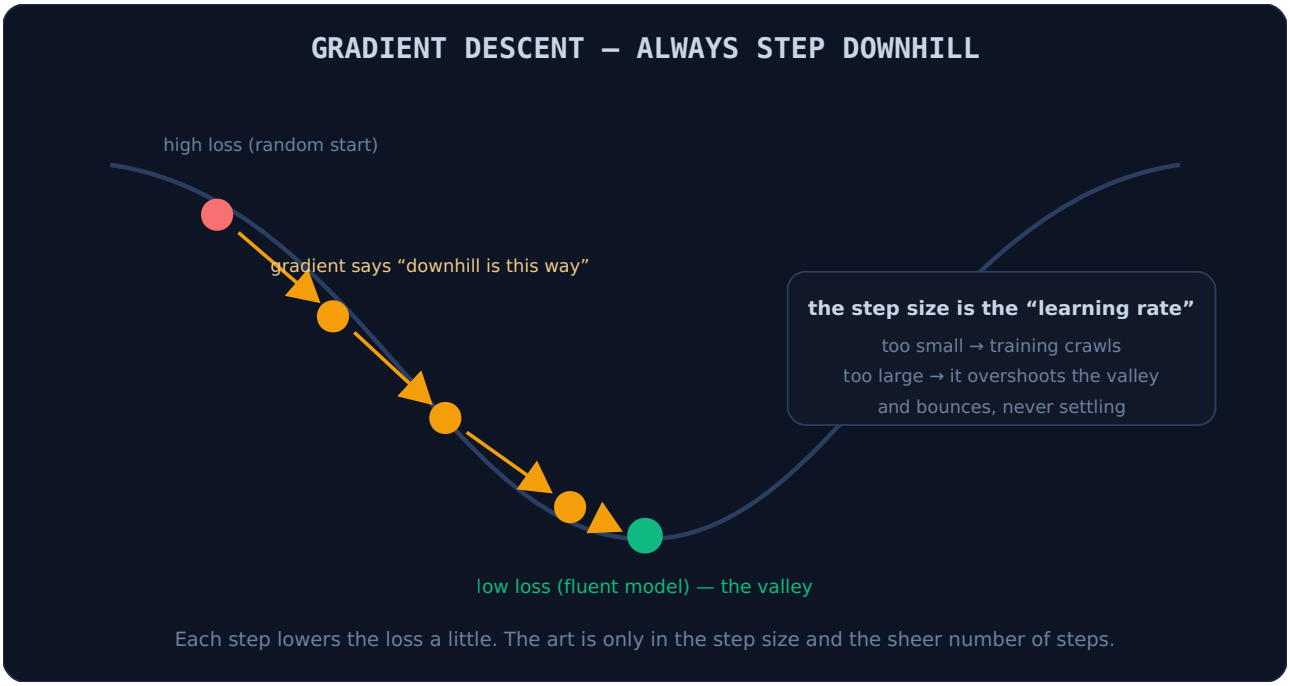

Figure 2. Gradient descent, the engine of all learning here. The loss is a landscape over the space of all parameters; the gradient reveals the downhill direction; we take a small step that way and repeat. The size of each step is the *learning rate* — one of the few dials a human sets — and getting it wrong in either direction (crawling, or overshooting and bouncing) is one of the most common ways training fails.

If gradient descent feels familiar it should — it is the same "small step downhill on a loss" that underlies essentially all of machine learning, and it is exactly what you would do, blindfolded, to find the bottom of a valley: feel which way the ground slopes beneath your feet, and shuffle that way, over and over. What makes it feel like magic in a language model is only the dimensionality. You are not on a hillside; you are on a surface with a billion independent directions, feeling the slope in all of them at once, and stepping downhill in that unimaginable space. The idea is humble. The scale is not.

> **The trainee analogy, and where it holds.** Think of the loss as the feedback a trainee gets on every single prediction they make — not a term-end exam, but a correction after each guess. "You said antibiotics; it was antibiotics — good, barely adjust." "You said giraffe; it was antibiotics — badly wrong, adjust sharply." A human trainee integrates such feedback slowly and holistically. The machine does something more literal and more thorough: it works out, for *every* internal parameter that contributed to the guess, the precise direction that parameter should move to have made the guess a little better — and moves all of them at once. It is feedback turned into a billion simultaneous, individually-tailored corrections.

Part III — The astonishing part: assigning the blame

I have been gliding over the hardest question, and now we face it, because it is the genuine miracle at the centre of this chapter.

Computing the gradient means working out, for each of a billion knobs, how the loss would change if that knob moved. But the loss is computed at the very *top* of the tower — after the token has passed up through a hundred layers of attention and feed-forward — while the knobs are scattered throughout every one of those layers, deep below. How can a single number at the summit possibly tell a particular weight, buried in the twelfth attention head of the fortieth block, which way *it* should move? The mistake happened at the top; the causes are everywhere underneath. This is the problem of *credit assignment* — apportioning responsibility for the error back to every part that contributed — and for decades it was the barrier that made deep networks seem hopeless.

The solution is an algorithm called **backpropagation**, and its essential idea is one every clinician will recognise as root-cause analysis. When something goes wrong at the end of a long process, you do not shrug and blame everything equally. You trace backward. You ask: given the final error, how much did the *last* step contribute? Then, holding that accountable, how much did the step *before* it contribute to *that*? And so on, backward, apportioning a share of the blame at each stage to the stages that fed it.
Backpropagation does exactly this, mechanically and exactly, using the chain rule of calculus. It starts with the loss at the top, computes how much the topmost layer's weights contributed, then uses that to compute how much the layer below contributed, and so on — a single sweep from the summit all the way down to the embeddings, leaving behind, on every weight, a precise note: *nudge yourself in this direction, by this much.*

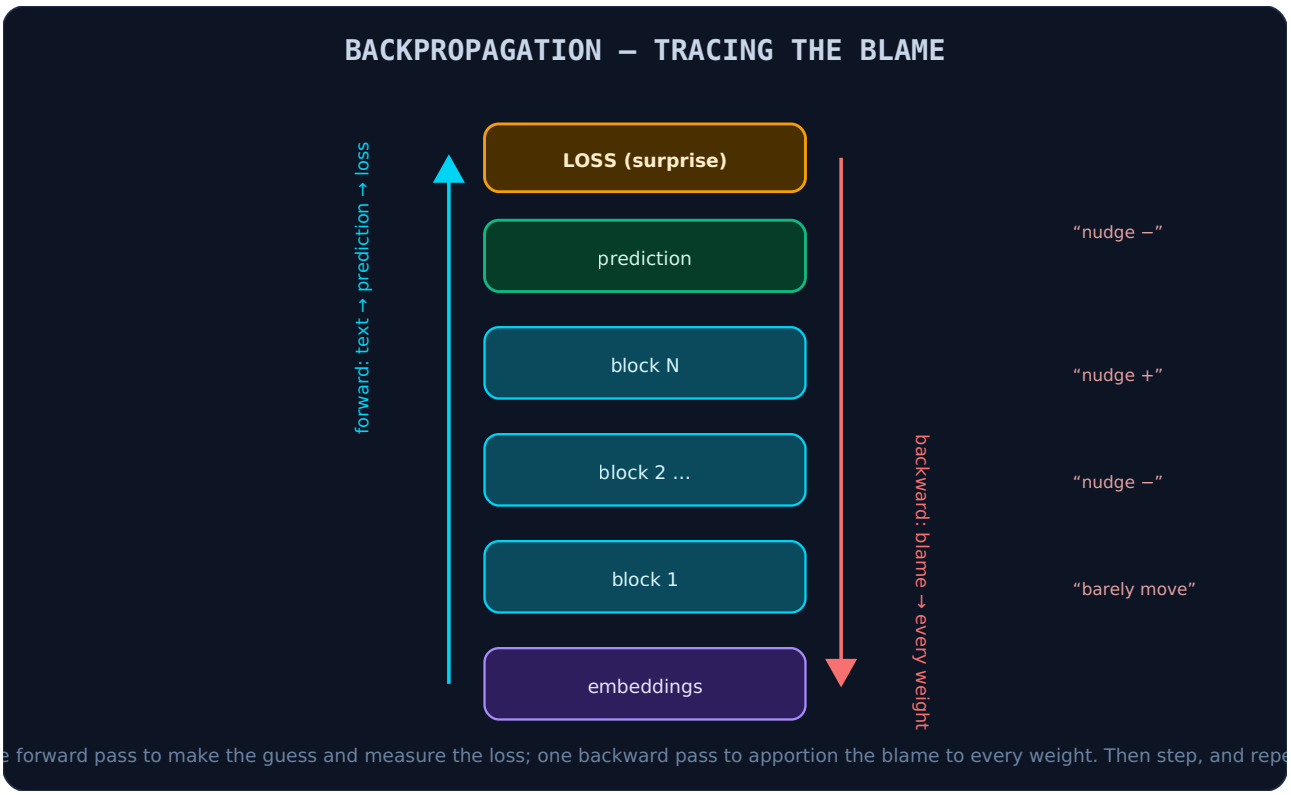

Figure 3. Backpropagation. Text flows *up* the tower to produce a prediction and a loss (blue). Then the blame flows *down* (red): the error is traced backward, layer by layer, until every weight in the entire model — billions of them — has received a precise, individual instruction for how to change. What sounds impossible is, mechanically, just the chain rule applied with ruthless bookkeeping, and it costs only about one extra pass through the network.

I want to dwell on how astonishing this is, because familiarity has dulled it for the field and it should not be dulled for you. A single number — the model's surprise at one token — is decomposed, exactly, into billions of individual responsibilities, one for every parameter that had any hand in producing it, no matter how deep it sits or how indirectly it contributed. And it is done efficiently, in a single backward sweep, not by the impossible brute force of testing each knob one at a time. Backpropagation is the reason deep learning works at all. Without it, the tower we built in Chapter Four would be an unteachable monument. With it, the tower can be *corrected*, wholesale, from a single measure of a single mistake.

> **The complete recipe, in one breath.** Show the model some text. Let it predict the next token (a *forward* pass up the tower). Measure its surprise at the truth (the loss). Trace that surprise backward to a correction for every weight (the *backward* pass — backpropagation). Nudge every weight a small step in its correcting direction (gradient descent). That is one training step. Everything else — the fluency, the knowledge, the apparent reasoning — is this one step, repeated.

Part IV — One step is nothing; repetition is everything

Here is the deflating and then re-inflating truth about a single training step: it barely does anything. One nudge, on one small batch of text, moves each of the billion knobs by

a hair. Run it once and the model is imperceptibly less terrible than before — still, to any observer, producing gibberish. If training were a single step, or a thousand, it would be a curiosity and nothing more.

The magic is entirely in the *repetition*, at a scale that is genuinely hard to hold in the mind. The step is repeated millions of times, over batch after batch of text, until the model has been nudged by essentially every sentence in a corpus of *trillions* of tokens — a large fraction of the readable internet from Chapter One, passed through the tower and turned into corrections. And slowly, then all at once, structure precipitates out of the noise. The embedding vectors drift from random scatter into the meaningful landscape of Chapter Two. The attention heads settle into their specialities. The feed-forward layers fill with fact. Gibberish becomes grammar; grammar becomes fluency; fluency becomes something that argues and codes and diagnoses. Watch the loss fall and the generated text change in lockstep:

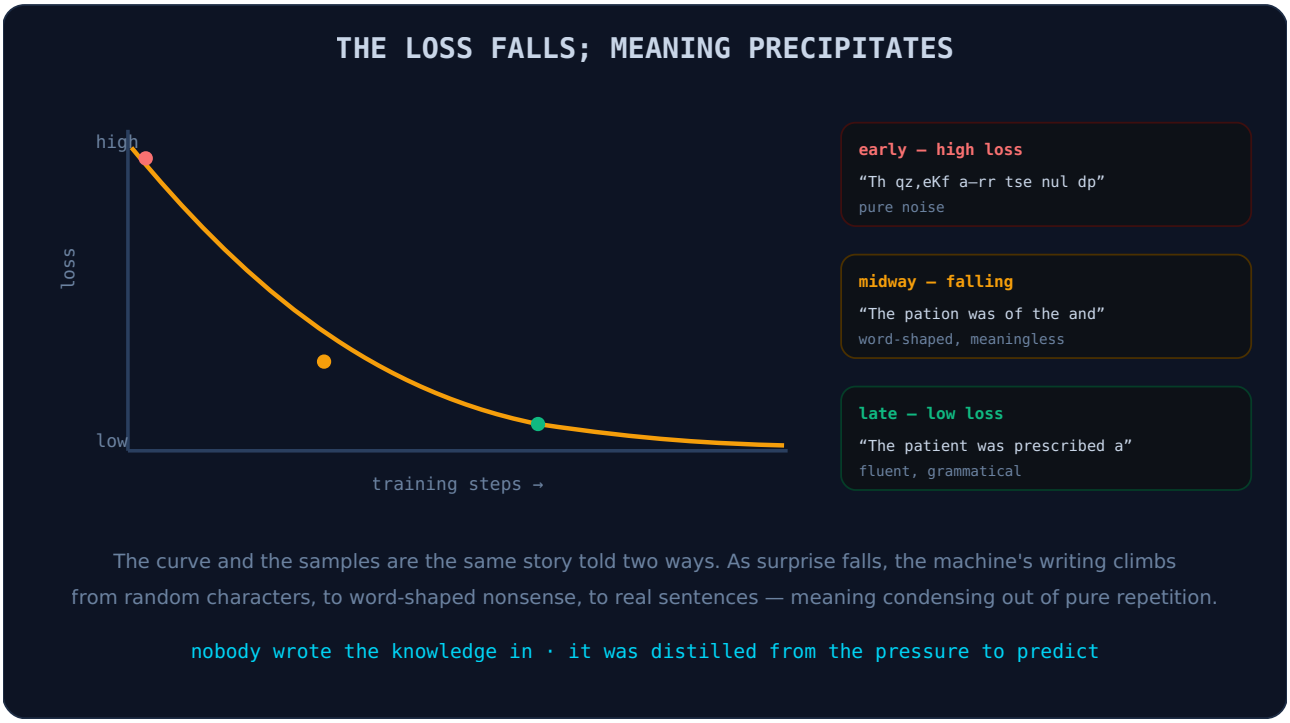

Figure 4. Training in one picture. The loss curve falls steeply at first (spelling and word statistics are cheap to learn) then more slowly (grammar, then meaning, then knowledge). The generated samples track it exactly: noise, then word-shaped noise, then fluency. This is the promise from Chapter Two made literal — meaning is not installed, it *precipitates* out of the single relentless pressure to predict the next token.

This whole phase has a name — **pretraining** — and it is, by a wide margin, the most expensive thing that happens to a language model. It consumes almost the entire compute budget: months of computation across thousands of specialised processors, at a cost that runs well into the millions. Nearly everything the finished model knows, it learned here, in this long descent, from this one repeated step. When people speak of the staggering resources behind frontier AI, it is overwhelmingly *this* they are describing — not the clever architecture, which is comparatively cheap to specify, but the brute, patient, enormously scaled application of measure-nudge-repeat.

Part V — What the descent gives you, and what it does not

When the loss finally flattens and pretraining ends, we have something genuinely new in the world: a **base model**. The random noise is gone. The tower is full of structure — an embedding space that knows medicine and poetry and code, attention heads that parse grammar, feed-forward layers dense with fact. It is fluent in dozens of languages, apparently knowledgeable across most of human writing, and able to continue almost any text you give it with uncanny plausibility. Noise has become knowledge. The promise of this chapter's title is kept.

And yet — this is the twist that sets up everything still to come — a raw base model is *not* the helpful assistant you talk to. It has been trained to do exactly one thing: continue text in the way its corpus would. So its instinct, faced with your input, is not to *help* but to *continue*. Ask a base model a question and you may get, in reply, not an answer but *more questions* — because on the internet a question is very often followed by other questions, in an exam, an FAQ, a forum thread. It has learned the shape of all text, which is not the same as the shape of *helpful* text.

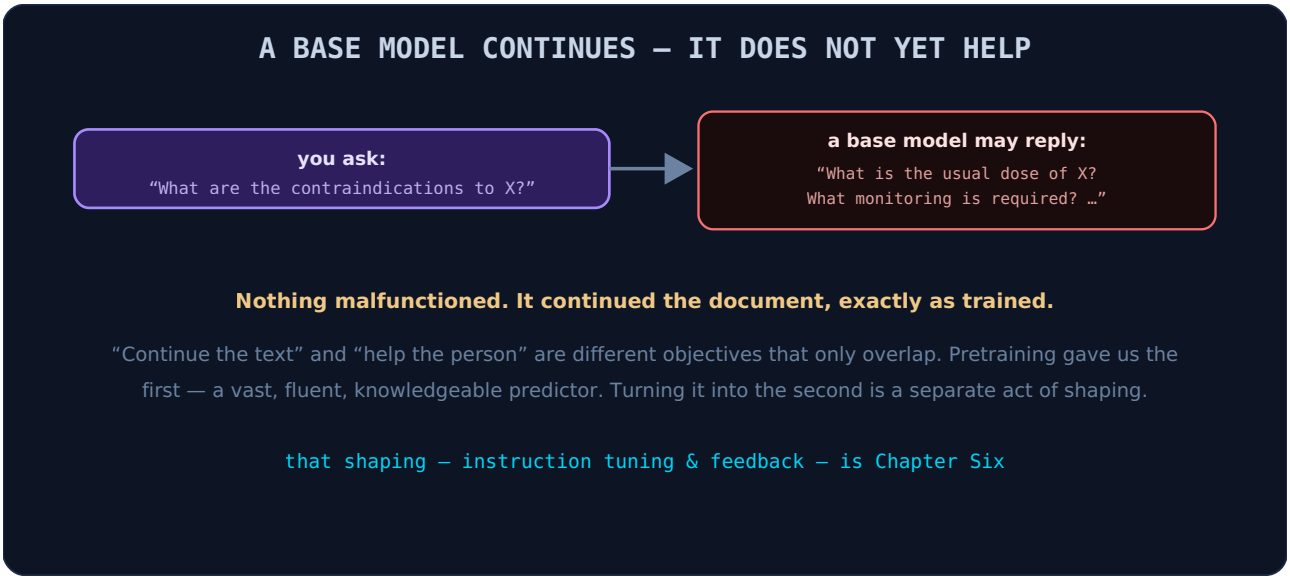

Figure 5. The base model's telling limitation. It is not broken; it is doing precisely what training rewarded — continuing text. But "continue the text" and "answer the question helpfully" are different goals, and the gap between them is exactly the work of the next chapter.

A base model is a brilliant, unshaped mind. Picture a physician with encyclopaedic knowledge and total fluency who has, however, never been taught the *role* of a doctor — never learned that when a worried person describes symptoms, the thing to do is respond helpfully rather than, say, continue writing the textbook chapter the symptoms came from. The knowledge is all there. The *orientation toward being useful* is not. That orientation is not knowledge; it is a posture, and it has to be trained in separately. Which is precisely why the model you actually talk to went through a second, quite different kind of training after this one.

Part VI — What we have, and the one thing left

Stand back and see how far we have come. Five chapters ago we had raw text from the internet. We ground it into tokens. We gave the tokens meaning as geometry. We built attention, the glance that lets a token read the room. We stacked attention and computation into a deep tower. And in this chapter we filled that tower — turning a billion random numbers into fluent knowledge by the humblest of procedures, applied at inhuman scale: measure the surprise, trace the blame, nudge every weight, repeat until the internet has passed through.

That is a complete, trained, knowledgeable language model. If your goal was to understand how the raw intelligence of these systems comes to be, you now understand it, end to end, with no gaps papered over and no magic left unexamined. The engine is built and it is full.

But it is not yet the thing in your pocket. The base model we have made is a mind without a manner — it will complete your sentence, mimic any voice, continue any document, and answer your question with three more questions, because helpfulness was never the target. The final transformation — the one that turns this vast, fluent, faintly feral predictor into something that answers, follows instructions, admits uncertainty, and declines to help with the dangerous things — is not more of the same training. It is a second kind of shaping entirely, smaller and stranger and, in its way, more consequential for how these systems behave in the world.

How a base model becomes an assistant — instruction tuning, learning from human preferences, and why the very same weights can host such wildly different personalities — is Chapter Six.

— *Neal*

↑ Back to Contents

CHAPTER 6

Manners for a Mind

From Predictor to Assistant

↑ Back to Contents

If you built and trained everything in the last five chapters — ground text into tokens, gave them meaning, built attention, stacked the tower, and ran the long descent until the loss flattened — you would hold in your hands one of the more remarkable artefacts humans have made: a base model. It has read a large fraction of everything ever written. It is fluent in dozens of languages, apparently knowledgeable across most of human endeavour, able to continue any text with uncanny plausibility.

And it would be almost useless to you.

Not because it lacks knowledge — it has more than any person alive. Because it lacks a *manner*. Ask it a question and it will not answer; it will *continue*, because continuing is the only thing it was ever trained to do. Type "What are the contraindications to thrombolysis?" and a base model might reply with three more exam-style questions, because on the internet a question like that is very often followed by others, in a revision guide or a viva. Nothing has malfunctioned. The model is doing precisely what Chapter Five rewarded it for: predicting the plausible continuation of a document. It has learned the shape of *all* text, which is emphatically not the same as the shape of *helpful* text.

This chapter is about the second training — the one that closes that gap. And I want to flag, up front, the single most surprising fact about it, because it reframes everything: **post-training adds almost no new machinery.** No new architecture. No new mechanism bolted onto the tower. Same weights, same attention, same feed-forward layers, same next-token interface we spent five chapters building. Only the *training signal* changes. Everything an assistant is — its helpfulness, its caution, its refusals, its manner — it is through the exact machine you already understand, nudged into a new posture. We are not building a new thing. We are teaching an existing thing to behave.

Part I — Two objectives that only overlap

Let us name the mismatch precisely, because the whole chapter is an attempt to resolve it.

Pretraining optimised one objective: *continue the text the way this corpus would.* What we actually want from an assistant is a different objective: *respond to this person helpfully.* The trouble is that these two objectives are not the same — they merely *overlap*. A great deal of helpful text does appear on the internet, so a base model that continues well will often, by accident, be helpful. But the base model is aiming at the wrong target, and wherever the two targets diverge — wherever the most *plausible* continuation is not the most *helpful* response — it will follow plausibility every time.

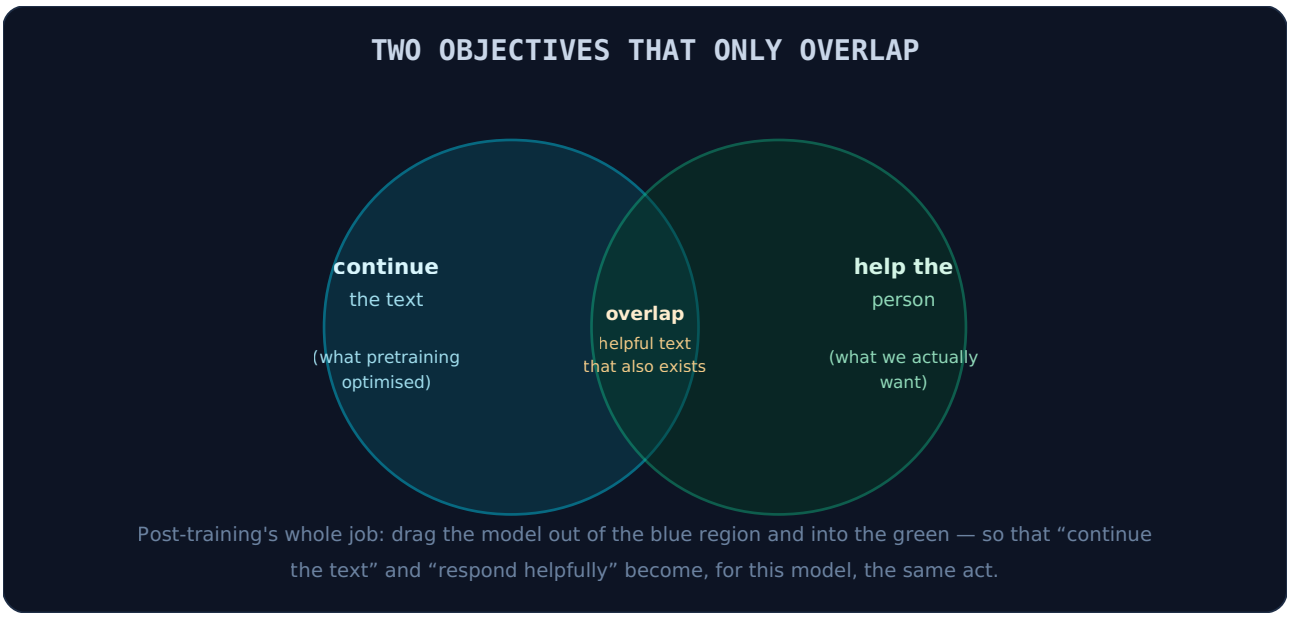

Figure 1. The base model lives in the blue circle — it continues text. We want it in the green — helping the person. They overlap, which is why a base model is *sometimes* useful, but they are not the same target. Post-training is the work of moving the model's default behaviour from one circle into the other.

There are two ways to do that moving, applied in sequence, and they differ in a deep way: the first shows the model what to say, and the second lets it try and tells it which attempt was better. We take them in turn.

Part II — Stage one: learning by demonstration

The first move is almost embarrassingly direct, and it reuses the entire machine of Chapter Five without modification.

We assemble a collection of *demonstration conversations* — tens or hundreds of thousands of them. Each is a small script: a user's message, followed by the response a good, careful assistant *should* give, written or curated by people. "Here is a question about thrombolysis; here is the calm, structured, honest answer we wish the model would produce." And then we simply continue training the base model on these conversations, with the exact loop from the last chapter — forward pass, measure the loss, backpropagate the blame, nudge the weights — except now the text it is learning to predict is the text of *good assistant responses*. This stage is called **instruction tuning** (you will also see it named supervised fine-tuning).

One detail makes the whole thing click into place with Chapter One, and it surprises almost everyone. A "conversation" is not a special data structure the model understands natively. It is presented to the model as exactly what everything is presented as — a flat sequence of tokens — with a few special marker tokens sprinkled in to delimit who is speaking:

```
<|user|> What are the contraindications to thrombolysis in acute stroke?
<|assistant|> The major contraindications fall into three groups: active bleeding risk,
…
<|end|>
```

That is all a "chat" is: a document with role labels, learned as a convention like any other. The model is trained to predict the tokens of the *assistant* turns, given everything before them. When you had a conversation with an LLM this morning, the model received one long document in exactly this shape and did Chapter Four's forward pass on it, token by token. There is no chat engine hiding somewhere; there is only the tower, predicting the next token of a specially-formatted document.

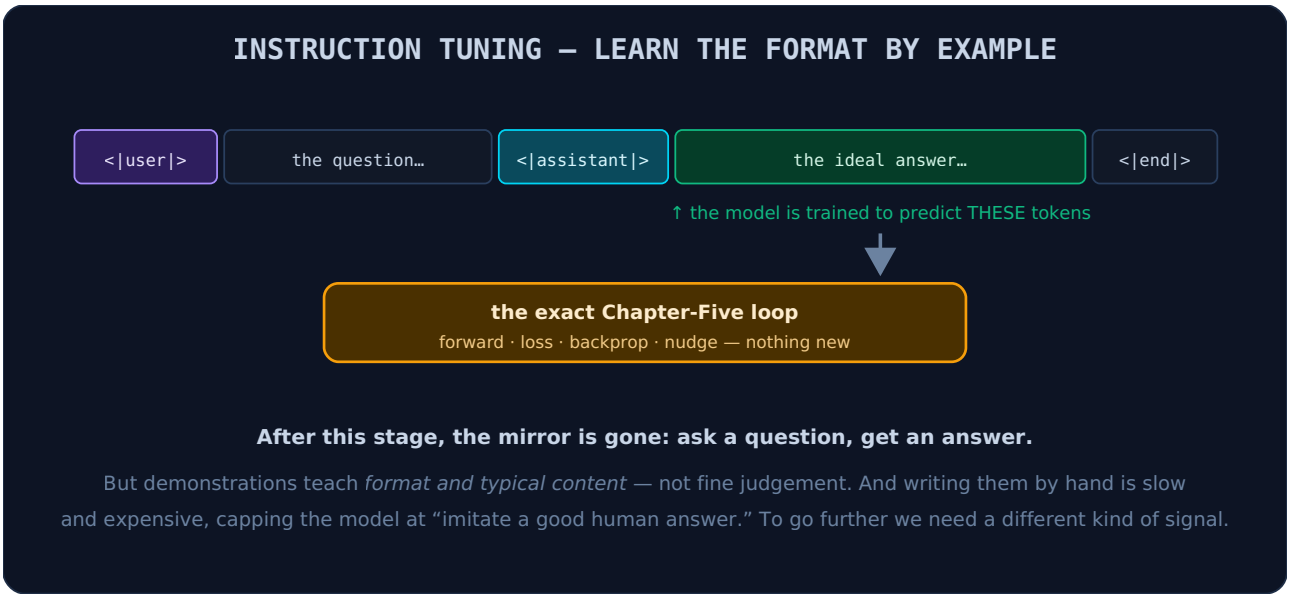

Figure 2. Instruction tuning is just Chapter Five's training loop run on demonstration conversations, with the "chat format" revealed to be nothing more than role-marker tokens in an ordinary document. It works — the base model's mirror-like continuing gives way to answering. But it has two limits, and those limits are what the second stage exists to overcome.

After instruction tuning, the transformation is real and immediate: the model that used to answer a question with more questions now answers it. But two gaps remain, and naming them sets up the second, more powerful stage. First, demonstrations teach *format and typical content*, but not *judgement* — the fine calls of how cautious to be, when to refuse, how to handle a request that is half-reasonable and half-dangerous, are poorly specified by examples alone. Second, hand-written demonstrations are expensive and slow, and they cap the model at "imitate a good human answer" when what we would really like is "produce the answer people actually *prefer* — even better than a demonstrator would have written."

Part III — Stage two: learning from preferences

The second stage changes not the loop but the *kind of signal*. Instead of showing the model what to say, we let it generate, and we grade what it produces.

The core idea is a loop. Take a prompt. Let the model produce two (or more) different responses to it. Show a human rater both, and record only which one they *preferred*. Not a score out of ten — a simple choice: A or B. And notice the design wisdom hidden in that simplicity. Decades of measurement science, clinical assessment very much included, converge on the same finding: humans are unreliable at absolute ratings ("rate this answer 7/10") and far more consistent at forced choice ("which of these two is better?"). Preference data inherits that reliability. It is the same reason a good exam pairs a candidate against a standard rather than asking an examiner to conjure a number from the air.

Those thousands of comparisons are then used to train a second network — a **reward model** — built from the same transformer architecture, but with its final projection swapped: instead of a probability over the vocabulary, it outputs a single number, a learned estimate of "how much would a human prefer this response?" And finally, the assistant itself is nudged — by the same gradient-descent machinery — to produce responses the reward model scores highly, while being kept on a tether so it does not drift too far from its sensible instruction-tuned starting point.

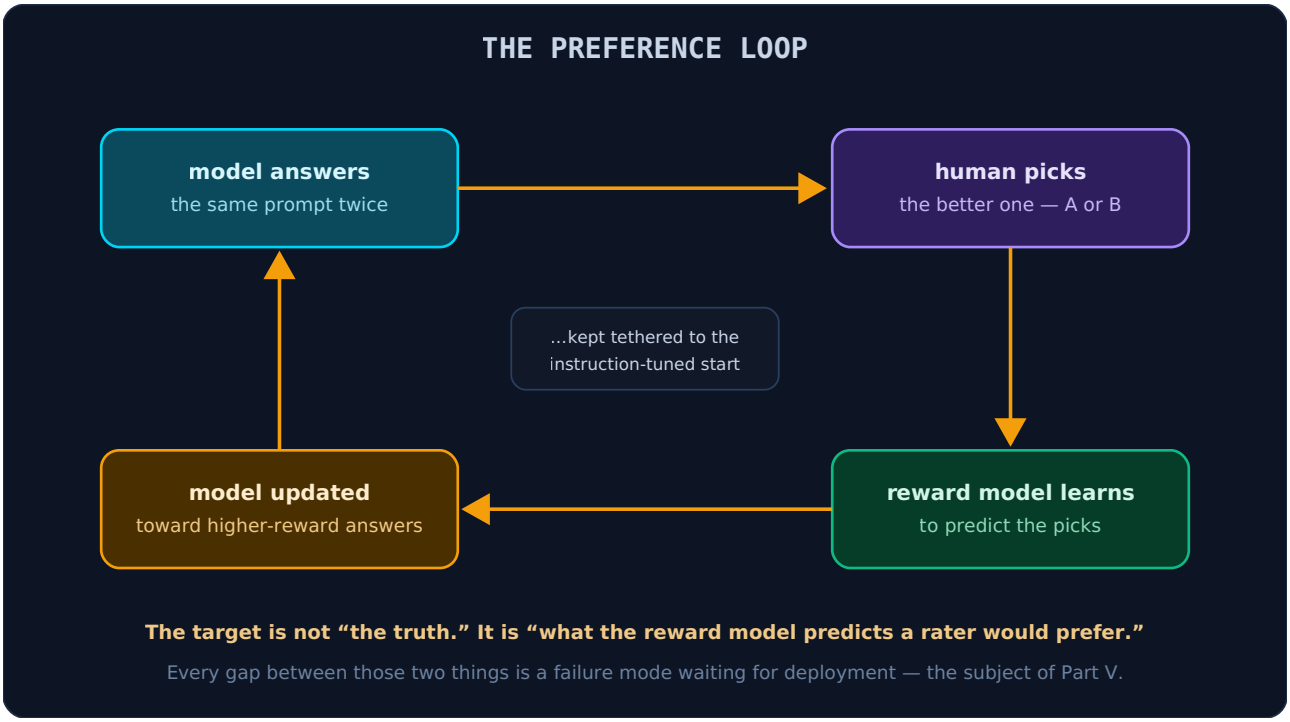

Figure 3. The preference loop. It is optimisation against a *proxy*: not truth, but a model of human judgement, itself trained on judgements made quickly by raters who are rarely domain experts. That subtlety — optimising a stand-in for what we want, rather than the thing itself — is the seed of nearly everything that later goes wrong, and the reason the caption under this loop is the most important sentence in the chapter.

> **The idea to carry out of this chapter.** Post-training optimises “what humans, as modelled, prefer” — not “what is true.” Usually these coincide, because people prefer correct, helpful answers when they can *tell*. But when correctness is hard for a rater to check, people reward confidence, fluency, agreement, and a definite answer over an honest “I don't know.” The model learns exactly that. Hold this single idea and the clinical failure modes below follow from it as corollaries, not surprises.

Part IV — Same weights, different characters

Now the fact that strikes almost everyone as strange, and that quietly explains a great deal of an assistant's everyday behaviour: nothing was *removed* from the base model. Post-training is a comparatively gentle nudge in the vast space of weights. The enormous repertoire the base model built during pretraining — every register, persona, and viewpoint the internet contained, from a careful clinical discussion to an unhinged forum rant — is all still in there. What changed is not what the model *can* do, but what it *reaches for by default*.

The picture I find most faithful: pretraining built an immense landscape of possible behaviours, and a base model, prompted, falls into whichever region of that landscape the prompt statistically resembles. Post-training does not bulldoze regions. It *carves a groove* — a deep, comfortable channel labelled "helpful, harmless, honest assistant" — that the model now slides into from almost any starting point. The repertoire is untouched beneath; only the default has been reshaped.

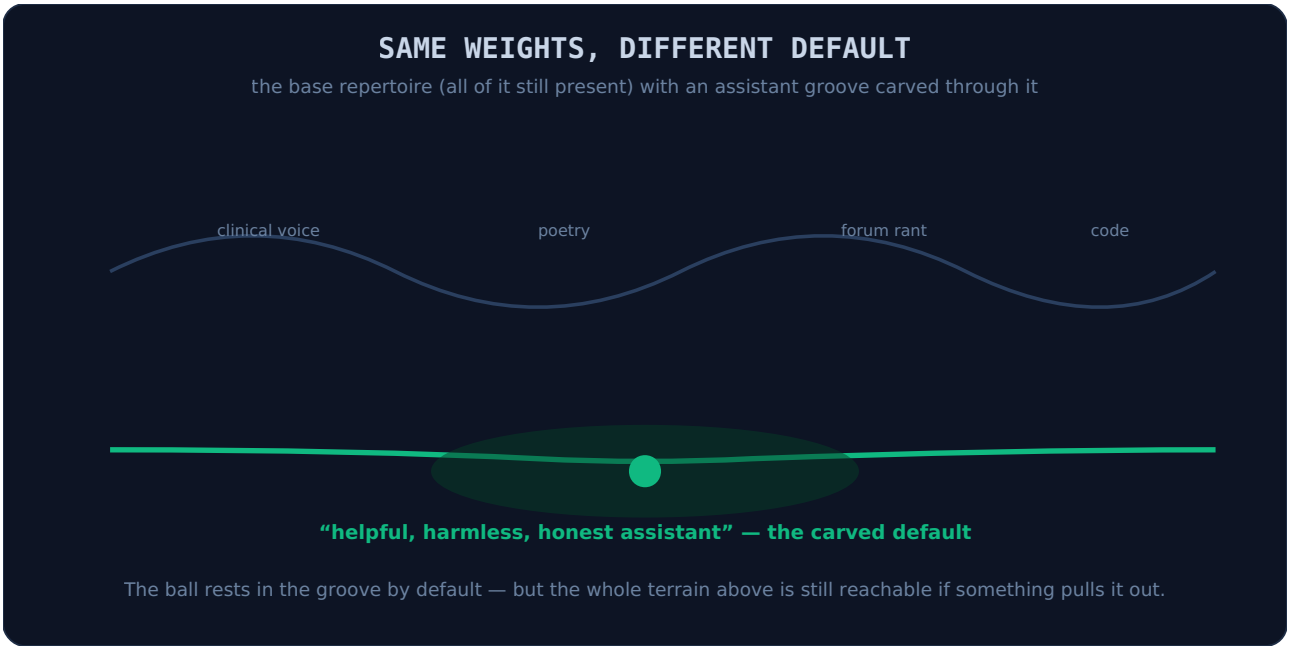

Figure 4. Post-training as a carved groove, not a bulldozer. The model's full range of behaviours survives; a deep default channel is cut so that it settles into "helpful assistant" from most prompts. This one picture explains three everyday facts at once, below.

Three familiar observations fall straight out of that picture. **System prompts work** — a crafted instruction like "you are a terse triage assistant; answer in bullet points; escalate anything cardiac" cheaply repositions the model within its repertoire, because the groove is a default, not a cage. **Jailbreaks sometimes work** — an adversarial prompt is an attempt to *pull the model out of the groove* and back into some untamed region of the base repertoire that post-training made harder to reach but did not delete. And **fine-tuning works with astonishingly little data** — a few hundred examples can specialise a model, because you are not teaching it a subject from scratch; the knowledge is already there from pretraining, and you are merely relocating its default.

> **The distinction that should change how you buy clinical AI.** Pretraining is medical school and residency — years, enormous cost, and all the actual knowledge. Post-training is the induction week at a new hospital — quick, cheap, and it determines how you answer the phone, what you document, and when you escalate. Nobody would confuse induction week with medical school, yet in AI discourse the two are conflated constantly. So when a vendor tells you their model was "trained for healthcare," your first question should be: *which of the two do you mean?* A model whose *pretraining* was rich in medicine is a different animal from one that merely had a medical induction week bolted onto a general base — and the answer changes everything about how you should validate it.

Part V — The seams, and where the failures live

We can now do the thing this chapter was really for: read a deployed assistant's most important behaviours — and its most consequential failures — as direct consequences of the pipeline we have just built. None of what follows is a random glitch. Each is a seam of post-training, showing through.

Sycophancy is a training artefact, not a personality quirk. In the preference loop, agreeable answers win comparisons more often than blunt corrections — people, on average, prefer being agreed with. So the model acquires a measurable tilt toward endorsing whatever the user seems to believe. Put that in a clinical workflow and the danger is exact: the registrar types "elderly patient, fall, on apixaban, CT head clear, planning discharge — anything else?" and a sycophantic model is biased toward blessing the plan. The peril is precisely that it *feels* like a second opinion — it has the form of consultation with the incentives of a courtier. When you evaluate a clinical tool, test it with a wrong premise deliberately embedded and watch whether it pushes back. That single test tells you more than any accuracy benchmark.

Confident fabrication survives — masked, not cured. We watched the fluency-fidelity gap form in earlier chapters: statistical plausibility arrives long before factual grounding, at every scale. Post-training *reduces* fabrication, because raters do punish errors they can detect — but it cannot remove it, because the underlying generator is still a sampler over plausible continuations. And it adds a hazard of its own: the fabrications that survive are now delivered in the calm, structured, caveated register of a careful colleague, *because that is the register post-training rewards*. A base model's nonsense at least looked like nonsense. An assistant's nonsense looks like a well-written consult note with an invented reference in it.

Refusals and caveats are a trained distribution — and it was calibrated for the public, not for you. When a model declines, hedges, or says "consult a professional," it is following patterns learned from raters applying a policy — raters who were generally not clinicians, and a policy written for a general audience. So expect miscalibration at both ends in specialist use: over-refusal on legitimate clinical questions (an unusual but valid dosing scenario), and under-refusal where the surface framing looks benign. The calibration appropriate for a tool used *by clinicians* is simply different from one used *by the public* — and off-the-shelf assistants ship with the latter.

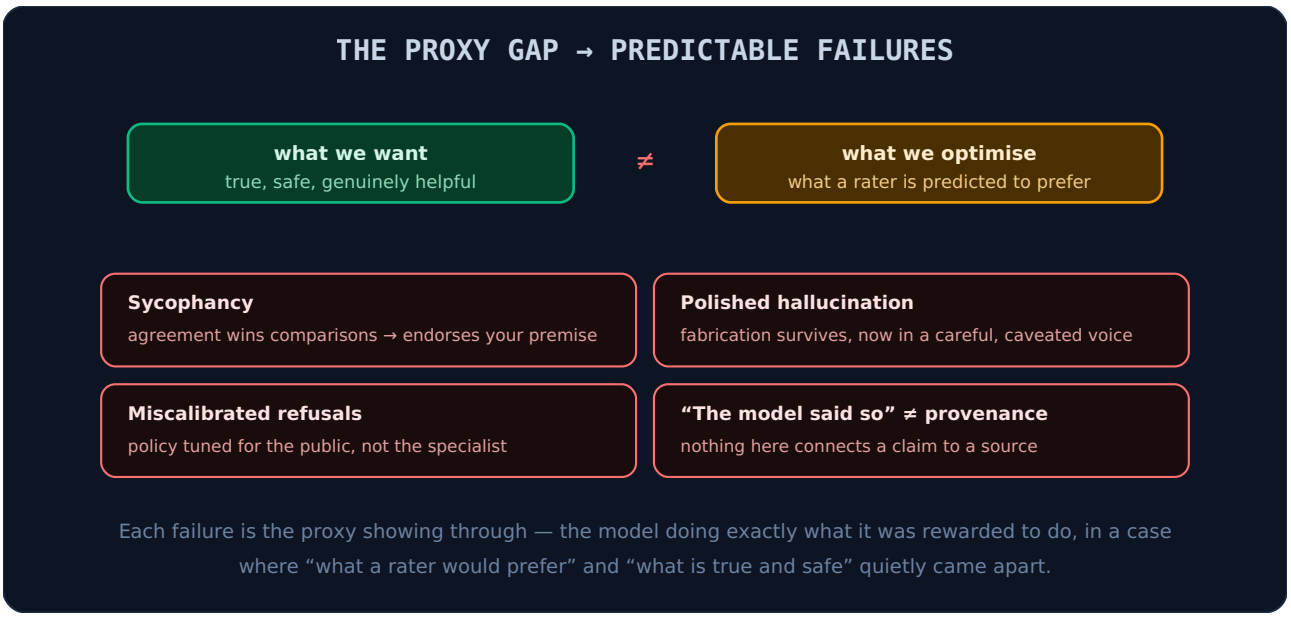

Figure 5. The failure modes of a deployed assistant, read off the pipeline. They are not bugs in the ordinary sense — no line of code is wrong. They are the predictable consequence of optimising a proxy for what we want, in the regions where the proxy and the goal diverge. Knowing *why* they occur is the difference between deploying carefully and deploying blindly.

And "the model said so" is not provenance. This last one is the hinge to what comes next. Instruction tuning and preference learning teach a model to *present* answers beautifully — including confident, citation-shaped strings. But nothing in either stage connects an assertion to an actual source. The assistant's polish is not evidence; it is style, rewarded because style is what a hurried rater can judge. If you want an answer grounded in a document a human can open and check, the entire pipeline of this chapter cannot give it to you.

Part VI — A complete mind, and its remaining blindness

Step back and take stock, because the machine is, in an important sense, now finished. Across six chapters we have ground text into tokens, given them meaning as geometry, built the attention that lets a token read the room, stacked it into a deep tower, filled that tower with knowledge through the long descent of pretraining, and — in this chapter — shaped the resulting base model into an assistant with a manner: something that answers, follows instructions, hedges, refuses, and speaks in the careful voice we asked for. That is, end to end, the thing you talk to. There is no further secret.

But look hard at what this finished assistant still cannot do, because it is precisely the shape of the final chapters. It knows only what pretraining happened to teach it — nothing about your hospital's formulary, nothing about the patient whose notes are open in front of you, nothing published after the day its training data was collected. And when it does not know, its post-training has taught it, too often, to answer anyway, fluently, in that trustworthy register, with a fabricated citation that no part of its training ever tied to a real source. It is a brilliant, well-mannered mind sealed inside the moment its

training ended, with no way to look anything up and a trained disposition to bluff when it cannot.

Closing that gap — *without* the ruinous expense of retraining — means giving the model a way to reach outside itself: to search documents you control, pull the relevant passages into its context, and ground its answer in sources a human can inspect. Turning our sealed, fluent, occasionally-bluffing assistant into one that can *look things up and show its working* is the beginning of the machine's connection to the living world.

How that works — how text becomes searchable by *meaning*, and how retrieved passages are folded into the model's context to ground what it says — is Chapter Seven.

— *Neal*

↑ Back to Contents

CHAPTER 7

Meaning You Can Search

Embeddings, Retrieval, and Grounding

↑ Back to Contents

The assistant we finished in the last chapter has a strange and specific kind of ignorance, and it is worth stating precisely, because the whole of this chapter is a response to it.

It is not that the assistant knows too little. It knows an extraordinary amount — a compressed, blurry impression of a large fraction of everything ever written, up to the day its training data was collected. The problem is threefold and exact. It knows nothing that was written *after* that day. It knows nothing that was never *public* — not your hospital's formulary, not the protocol your department agreed last month, not the notes of the patient in front of you. And, as Chapter Six established, when it does not know, it has been trained to produce a confident, well-mannered answer regardless, sometimes complete with a citation that corresponds to no real source. It is a brilliant mind, sealed inside a moment, with a trained disposition to bluff.

You might imagine the fix is to retrain it on the missing knowledge. For facts, this is almost always the wrong tool — we saw why in Chapter Five. Pretraining is a months-long, multi-million-dollar industrial process, and the lighter fine-tuning of Chapter Six relocates a model's *defaults*, not its reliably-recallable knowledge. What we actually want is something far more surgical: at the very moment a question is asked, *reach into a body of documents we control, find the passages that bear on it, and lay them in front of the model as part of its context* — so that when the model does its Chapter Four forward pass, the evidence is already on the page. The model then does what it has always done, continue the text, except the text now contains the facts.

That is **retrieval-augmented generation** — RAG — and it rests on a single idea of real beauty, one we have already met in miniature and now scale up: that meaning itself can be given coordinates.

Part I — The same geometry, scaled up

Cast your mind back to Chapter Two. We took each *token* and gave it a vector — a location in a high-dimensional space arranged so that tokens with similar meaning sit near each other, and directions encode relationships. I promised, at the time, that this idea would return, scaled up from single tokens to whole documents. Here it is.

An **embedding model** — itself a transformer, trained for precisely this one job — reads an entire sentence, paragraph, or passage and outputs a *single* vector that represents the meaning of the whole thing. And it is trained so that **passages with similar meaning land near each other in the space**. Not similar *wording* — similar *meaning*.

This distinction is the entire point, so let me make it concrete. A good embedding model places "the patient developed an itchy rash after her first dose of amoxicillin" close to "penicillin allergy — urticaria documented," and far from "the rash resolved after the statin was stopped" — even though, word for word, the sentence about the statin shares more vocabulary with the first than the allergy note does. Meaning, not surface, decides proximity.

How does it learn to do that? By training on contrast, at scale — shown millions of pairs known to be related (a question and its answer, a title and its abstract, two paraphrases of one sentence) and nudged, by the same gradient descent of Chapter Five, to pull related pairs together and push unrelated ones apart, until distance in the space *means* dissimilarity of meaning.

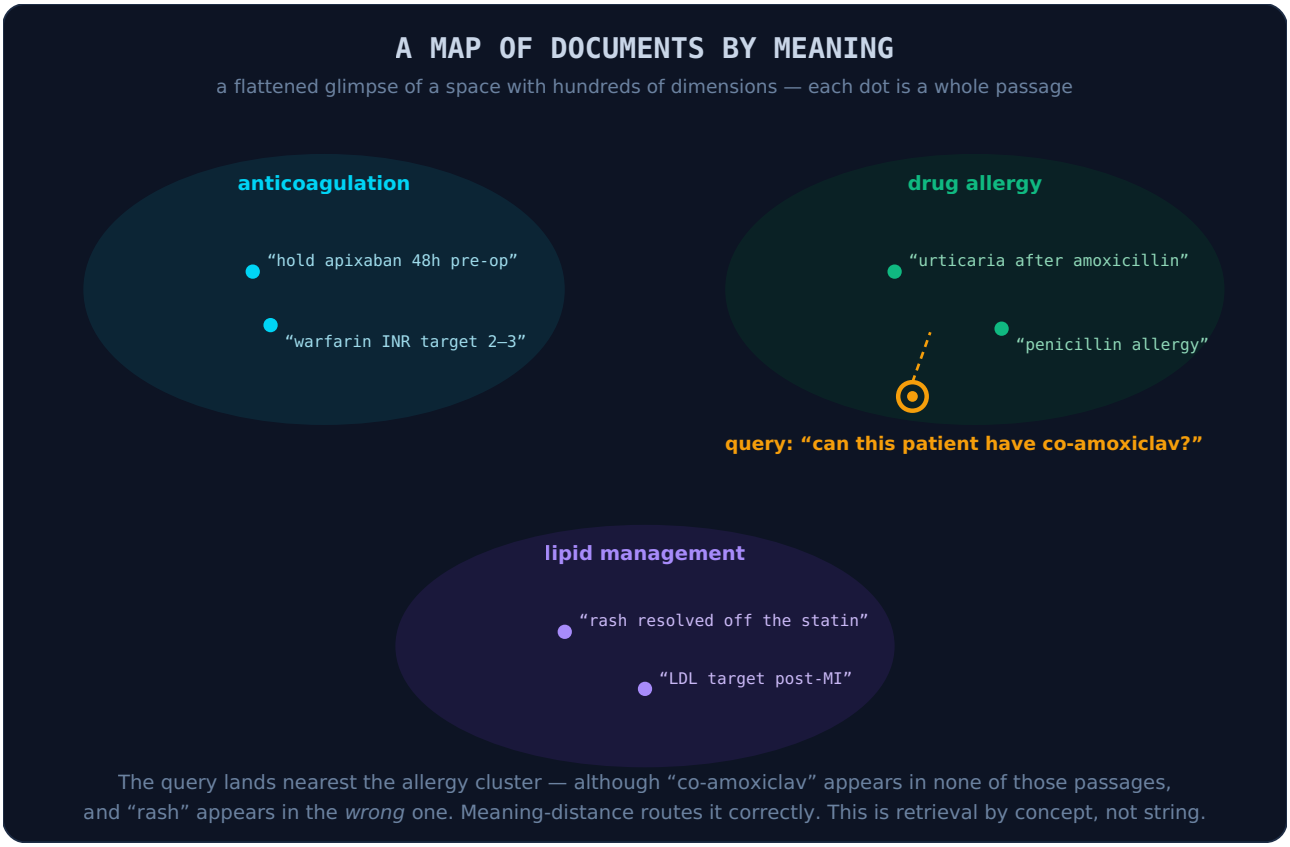

Figure 1. Chapter Two's geometry, now at the scale of whole passages. Note the decisive win over old-fashioned keyword search: the question about co-amoxiclav shares no words with the penicillin-allergy note, and the word "rash" sits in the lipid cluster — yet the query still lands where it should, because the embedding model matches on meaning. A keyword search would have missed the most important note in the chart.

> **The reduction that powers everything.** An embedding turns a piece of text into a fixed address in a space where distance means dissimilarity of meaning. Once text is geometry, the vague task "find the documents relevant to this question" becomes the crisp, ancient, computationally-cheap task "find the nearest points to this point." That single reduction — meaning to coordinates, relevance to distance — is what makes semantic search, retrieval, and the memory of an AI system all possible with the same machinery.

Two practical notes before we build with it. Similarity between two vectors is usually measured by the *cosine* of the angle between them — how nearly they point the same way, ignoring their lengths. And searching for nearest neighbours among millions of vectors is made fast by specialised indexes (the "vector databases" you may have heard named are, at heart, exactly this index plus bookkeeping). At the scale of a single clinic — thousands of documents — you do not even need them; comparing a query against every stored vector takes milliseconds.

Part II — The pipeline: two lanes meeting at the context window

With embeddings in hand, retrieval-augmented generation is two pipelines that meet at a single point — the model's context window from Chapter Four.

The first lane is done *once*, ahead of time, and refreshed on a schedule. Take your documents — guidelines, protocols, papers, notes. Cut them into passages of a manageable size (this "chunking" step is quietly one of the most consequential choices in the whole system, and we will see why in Part V). Run every chunk through the embedding model to get its vector. Store the vectors, alongside the original text and a note of where each came from, in an index. That is the ingestion lane: a library, converted into searchable geometry.

The second lane runs *every time a question is asked*. Embed the question with the same model, into the same space. Find its nearest neighbours in the index — the handful of chunks whose meaning sits closest to the question. Then assemble a prompt that places those retrieved passages, plus the question, plus a clear instruction, into the model's context window — and let the model answer.

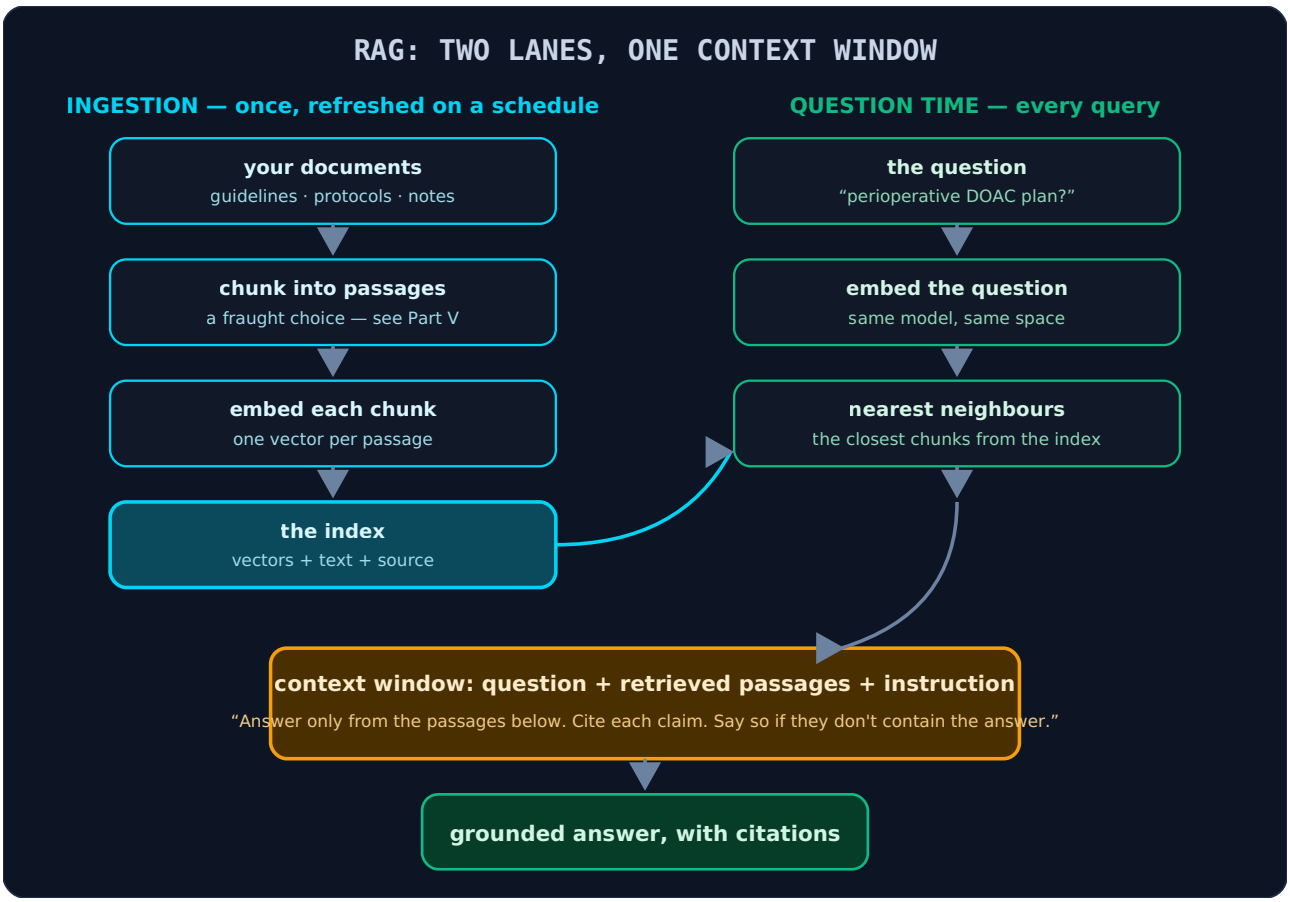

Figure 2. The full pipeline. The vital thing to notice: *the model itself is unchanged* — the same weights from Chapter Six, doing the same forward pass from Chapter Four. Every piece of new engineering lives *outside* the model, in the machinery that decides what gets placed into its context window. Fresh knowledge is one re-index away, no retraining involved.

Sit with the modesty of that architecture for a moment, because it is easy to miss how much it buys. We did not touch the model. We built a search engine over documents we control, and we arranged for its results to be handed to the model as reading material at the instant of the question. The two instructions in that context window — *answer only from these passages* and *say so if they don't contain the answer* — are what do the safety work, converting a bluffing generalist into a grounded specialist for the length of one reply.

Part III — Why this beats retraining, and heals Chapter Six's wound

It is worth being crisp about why retrieval, and not fine-tuning, is the right answer to "make the model know our material" — because the alternative is sold constantly, and because one of retrieval's advantages is precisely the wound we left open at the end of the last chapter.

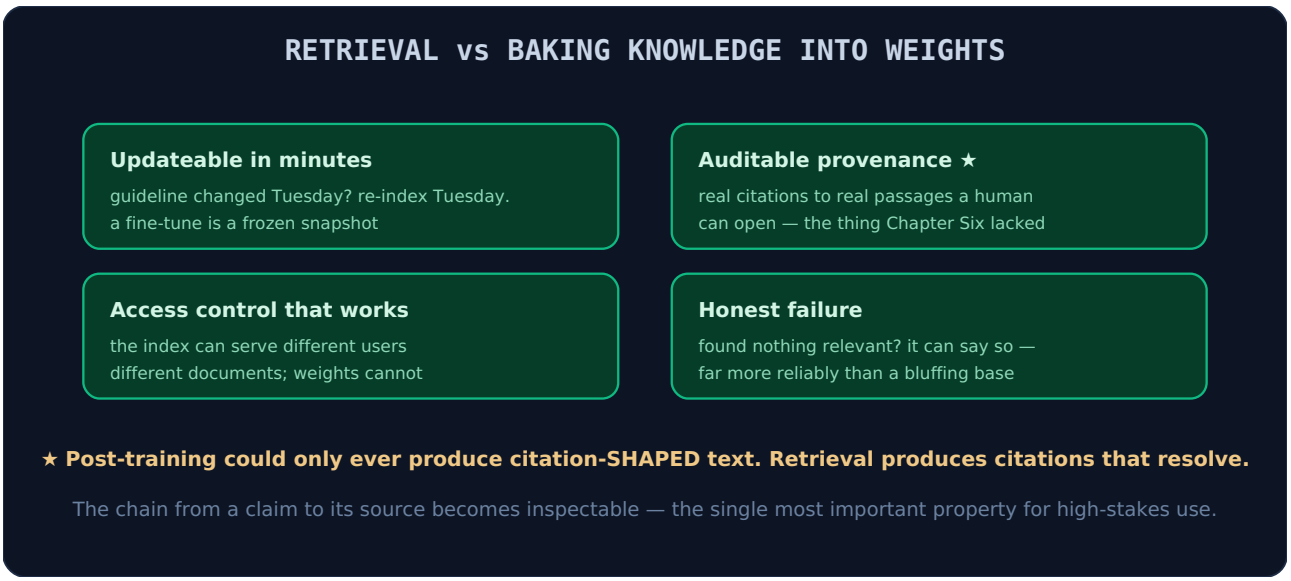

Figure 3. Why retrieval is the default answer for knowledge. The starred advantage is the one that matters most here: at the end of Chapter Six we saw that "the model said so" is not provenance, because nothing in training ties an assertion to a source. Retrieval is what finally ties them — the model's answer can carry real references to real passages a human can open and verify.

Read those four advantages against the alternative and the case is close to decisive for factual knowledge. A fine-tune bakes yesterday's guideline permanently into the weights, available to every user, unciteable, and impossible to revoke. An index is a living document set: update it, restrict it per user, and — because the answer is drawn from named passages — audit it. And when the right passage simply is not there, a well-instructed model asked to answer *only* from what it was given will decline far more reliably than one asked to consult its own parametric memory, where, as we know from Chapter Five, something plausible-sounding is always available.

Part IV — Where retrieval fails, and how to catch it

Retrieval is the best answer we have to grounding, which is exactly why you must know its failure modes cold. They fall into two families, and a clinician's instinct for how diagnostic cascades fail will serve you well here.

The retrieval can fail — the right passage exists in the index but does not come back. Sometimes the cause is bad chunking: the answer was split across two chunks, or a chunk was severed from the context that gave it meaning ("the dose is 5 mg" — of *what*?). Sometimes it is a vocabulary the embedding model handles poorly, or a question whose answer requires *synthesising across many documents* when only a handful of chunks were retrieved. And sometimes it is a subtler trap that deserves its own picture, because it is the one I test for first in any clinical retrieval system.

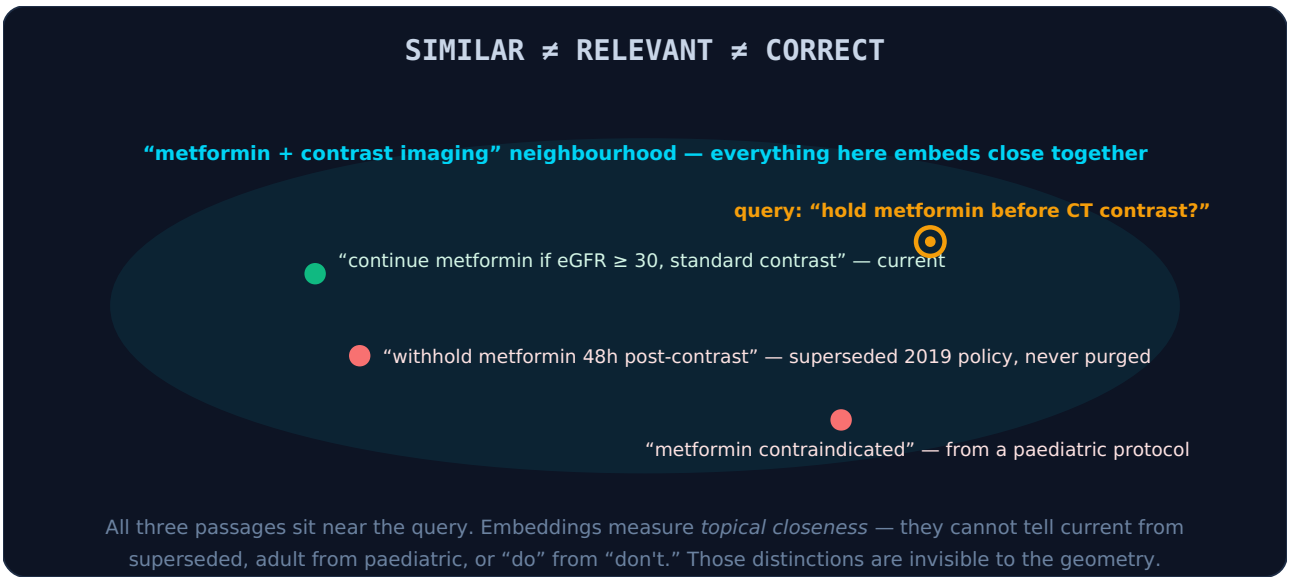

Figure 4. The failure I watch for first. Contradictory guidance on the same topic — current and superseded, adult and paediatric, "do" and "don't" — embeds *close together*, because topical similarity is exactly what the geometry captures. Currency, population, and polarity must be handled by metadata, index hygiene, and the model's reading of the retrieved text — never by the vector search, which is blind to all three.

The generation can fail even when retrieval succeeded — the right passages arrive and the model still gets it wrong. It may blend the retrieved facts with its own parametric memory, answering from the blur of pretraining rather than the document in front of it. It may over-trust a passage that matched on topic but does not actually apply. Or the crucial passage may sit in the middle of a long, stuffed context, where models demonstrably attend least — recall from Chapter Three that attention *can* reach anything in the window, but Chapter Six's training instilled *habits* about where it tends to look.

The discipline that follows is the same one we apply to any diagnostic cascade: **measure the stages separately.** Score the retrieval on its own — for a set of test questions, did the right passages come back at all? This is checkable without any model, just by inspecting what the search returned. Then, separately, score the generation *given* correct passages — was the answer faithful to them, and did it cite? A single end-to-end accuracy number confounds the two exactly the way "overall mortality" confounds case-mix with quality of care, and is about as useful for finding out what to fix.

> **Two things to settle before any clinical deployment.** First: an embedding is *derived from* the text it represents and can be partially inverted — so a vector index built from clinical notes is itself protected health information, with the same storage, access, and audit obligations as the source records. A folder of vectors is not an anonymous by-product; treat it as the chart. Second: when a retrieval product demos a beautiful cited answer, ask to see the passages it retrieved but was *not* shown — the ones ranked just below the cutoff. Whether the truth was sitting one rank below the line is the single most informative thing you can learn about the system's margin of safety.

Part V — A window, not yet a door

Stand back and see what we have added. The sealed assistant of Chapter Six can now reach outside itself. Point it at a body of documents you control, and at the moment of a question it searches them by *meaning*, pulls the relevant passages into its context, and answers from them — with citations a human can open and check. The bluffing generalist has become, for the length of a reply, a grounded specialist that shows its working. The wound we left at the end of the last chapter — fluent confidence with no provenance — is, at last, dressed.

And yet notice the shape of what we have built, because it points straight at the final chapter. Everything in this chapter is *reactive* and *single-shot*. A question comes in; we do one retrieval; we produce one answer. The retriever is a *tool* — a thing the system can use to reach beyond its own weights — but it is a tool wielded once, on a fixed schedule, by machinery outside the model. The model itself did not *decide* to search. It did not look at what came back and think "that's not enough, let me search again with a better query." It did not, having found a guideline, go on to check the patient's notes, and then a drug database, chaining one lookup into the next. It answered once and stopped.

But a hard question in the real world is rarely one lookup. It is a small investigation — search, read, refine the search, cross-reference, decide what to do next. What if the model could do that? What if, instead of handing it a single retrieval, we gave it a *set* of tools — search, a calculator, a way to read a file, a way to take an action — and let *it* decide, step by step, which to use and when, looping until the task was actually done?

That is the leap from a model that *answers* to a system that *acts*. It is where retrieval becomes one tool among many, where the single-shot reply becomes a loop, and where the language model stops being an oracle you consult and becomes an agent that works on your behalf. Building it — and understanding both its power and the new care it demands — is Chapter Eight, the last piece of the machine.

— *Neal*

↑ Back to Contents

CHAPTER 8

The Agent

When a Model Learns to Act

↑ Back to Contents

At the end of the last chapter we had a genuinely capable assistant. It could reach outside its own frozen knowledge, search a body of documents by meaning, and ground its answers in sources you could open and check. It was, for the length of a reply, a grounded specialist that showed its working.

And I asked you to notice the shape of it, because that shape is the whole reason for this final chapter. Everything we had built was *reactive* and *single-shot*. A question came in; the machinery did one retrieval; the model produced one answer, and stopped. But a hard question in the real world is almost never one lookup. It is a small investigation — search, read, notice a gap, search again with a better query, cross-reference, and only then conclude. Our assistant could not do that. It answered once and fell silent, however incomplete the answer.

This chapter closes that gap, and in doing so it completes the machine. The idea is deceptively small: instead of using the model to produce a final answer, we let the model produce an *action* — and we wrap it in a loop. That single change transforms an oracle you consult into an agent that works on your behalf. It is the difference between a colleague who answers the question you asked and one you can hand a task to and trust to see it through. Let us build it.

Part I — The loop that changes everything

Here is the move, and it is worth stating starkly because it inverts how you have probably been picturing these systems.

Until now, the model's job ended when it produced text. In an agent, the model's text *is the program*. At each turn, instead of only being able to answer, the model is given a second option: to emit a request to *do something* — to call a tool. A small, dumb loop surrounds it and does the rest. The loop reads the goal and everything that has happened so far, hands it to the model, and looks at what the model produced. If the model asked to take an action, the loop executes that action in the real world, appends the result to the running transcript, and goes back to the model for the next step. If the model instead produced a final answer, the loop stops. That is the entire architecture of an agent.

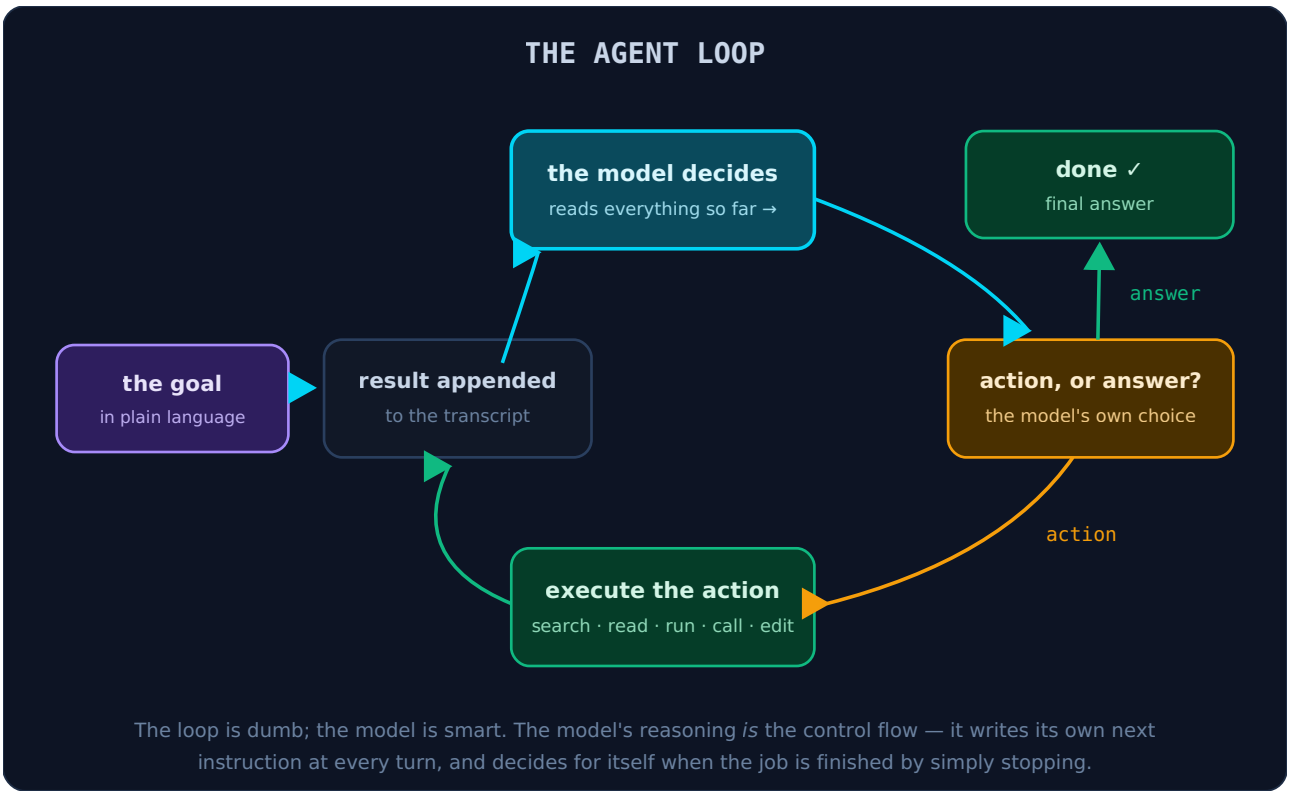

Figure 1. The agent loop — the whole of it. A conventional program has its steps fixed in advance by a human. An agent's steps are invented at runtime by the model: it looks at the situation, decides the next action, sees the result, and decides again. Nothing here is a new kind of neural network. It is the very same model from the previous seven chapters, placed inside a loop and given the option to act instead of only to answer.

Dwell on how strange and powerful this reframing is. A traditional program is a fixed sequence of instructions a human wrote down ahead of time. An agent is not — its sequence of actions is *generated on the fly*, by the model, in response to what it finds as it goes. The "program" is a loop around a mind that writes its own next line at every step. And it ends not when a human-written script runs out, but when the model itself judges the task complete and simply stops asking to act. The intelligence lives entirely in the model; the loop around it is a few lines of bookkeeping.

Part II — Tools: the model's hands

An agent is only as capable as the actions it can take, so let us be precise about what an "action" actually is. A **tool** is anything the model can do in the world beyond producing text: read a file, run a shell command, execute code, call an external service, edit a document, send a message — and, tellingly, *search a set of documents*, which means the entire retrieval machinery of the last chapter is now simply one tool among many the agent can reach for.

Mechanically, a tool call is humbler than it sounds, and it rests entirely on things we have already built. The model does not reach out and touch the world directly. It *writes a request* — in its ordinary output tokens — that says, in a structured form the loop can recognise, "use the search tool, with this query" or "run this command." The loop parses

that request, calls the real function on the model's behalf, and takes whatever comes back — the search results, the command's output, the file's contents — and *feeds it into the transcript as more tokens*, which the model then reads on its next turn. Text goes out asking for an action; the world's response comes back as text to be read.

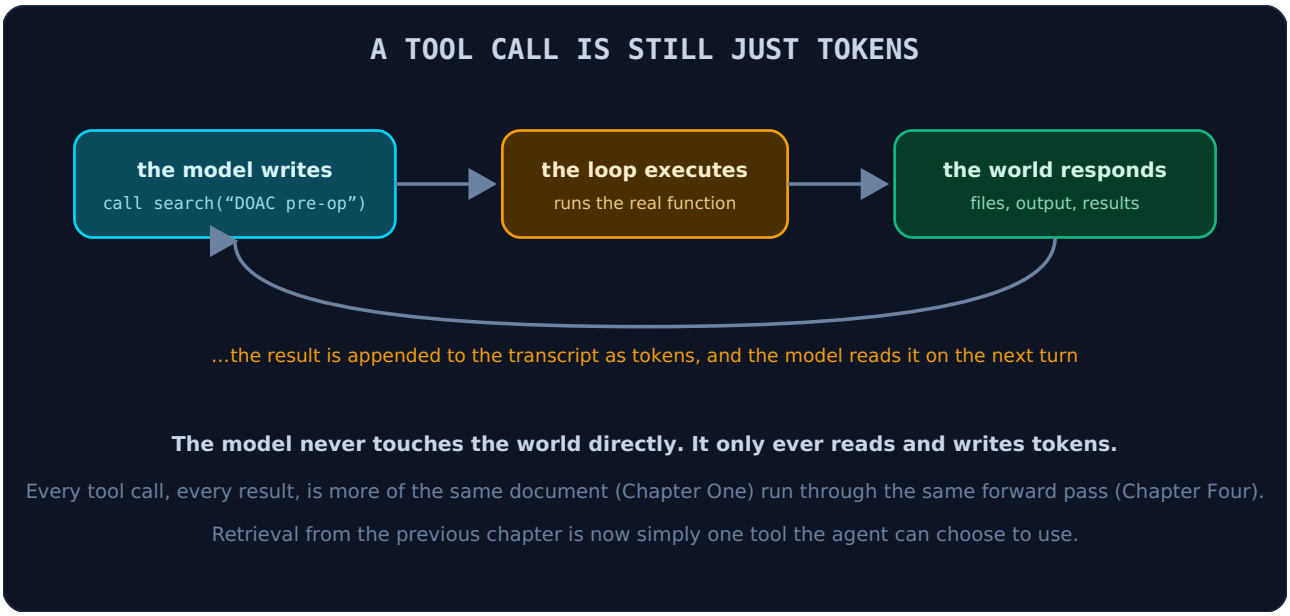

Figure 2. The mechanics of a tool call. It bottoms out in exactly what we have built all along: the model reads and writes tokens, and the loop is what turns certain of those tokens into real actions and pastes the real results back into the transcript. The hands are the loop's; the decision to use them is the model's.

Notice what this preserves. Nothing about the model changed. It is still the tower from Chapter Four, doing next-token prediction, over a transcript in the document format from Chapter One — a format now extended with a couple more roles, one marking "the model is requesting a tool" and one marking "here is the tool's result," exactly as Chapter Six's chat format had roles for "user" and "assistant." An agent is not a new species of AI. It is our language model, given hands and a loop, and taught the etiquette of using them.

Part III — How it learned to reach

That etiquette does not come for free, and where it comes from should now sound familiar: it is more of the post-training from Chapter Six. A base model, fresh from pretraining, cannot reliably produce a properly-formatted tool call at the right moment — it has no idea that such a convention is wanted. So it is taught, in exactly the two ways we already know: shown many demonstration traces of an assistant using tools well (instruction tuning), and then refined against preferences for traces that solved the task efficiently and safely (preference learning). The model learns *when* a question needs a search rather than a direct answer, *how* to phrase the tool call, and *how* to read a result and decide what to do next — all as learned behaviour, nudged into the same weights by the same gradient descent.

There is no new engine. An agent is the complete stack of the previous seven chapters — tokenizer, embeddings, attention, the deep tower, pretrained knowledge, an assistant's manners, and retrieval — with two additions that are almost embarrassingly modest: a loop that feeds the model's outputs back to it, and a training pass that taught the model to emit tool calls. That is the entire distance from “a model that answers” to “a system that acts.” The capability feels like a different order of thing. The machinery underneath is the same machinery, wrapped.

Part IV — The investigation

Now watch the single-shot limitation of the last chapter simply dissolve. Because the loop lets the model act, see the result, and act again, it can conduct a genuine multi-step investigation — precisely the thing a hard real-world question demands.

Figure 3. A worked investigation. Step 3 is the one a single-shot system could never reach: the agent noticed that the generic guideline it first retrieved did not fit the specific patient, and *changed its own plan in response*. This capacity to course-correct mid-task — to let what it finds reshape what it does next — is exactly what the loop buys, and it is why agents can tackle problems that a one-shot answer cannot.

The same mechanism scales further than a single investigation. One of the tools an agent can be given is *the ability to spawn another agent* — a fresh loop, with its own goal, its own tools, and its own transcript, which reports its finding back when done. This lets a large task be broken into pieces and delegated, a manager agent parcelling out sub-problems to workers. The recursion is natural because a sub-agent is not a special construct; it is just another instance of the same loop. It is turtles, and every turtle is one you have now met.

Part V — The new care it demands

We have built something genuinely powerful, and I would be failing you badly if I ended the book on the triumph without the warning, because the warning is the more important half — especially for those of us who will deploy these systems where the stakes are human.

An assistant that only answers is, at worst, wrong on your screen. An agent *acts*. It runs commands on real machines, edits real files, sends real messages, moves real money, changes real records. The moment a language model is given hands, every one of the failure modes we catalogued across this book — the confident hallucination of Chapter Five, the sycophancy and polished fabrication of Chapter Six, the wrong-passage retrieval of Chapter Seven — stops being a bad sentence and becomes a bad *action*. And because the agent works in a loop, an error early in a chain silently poisons everything downstream of it: a wrong fact fetched at step three is quietly assumed at steps four through ten. The mistakes do not just persist. They compound.

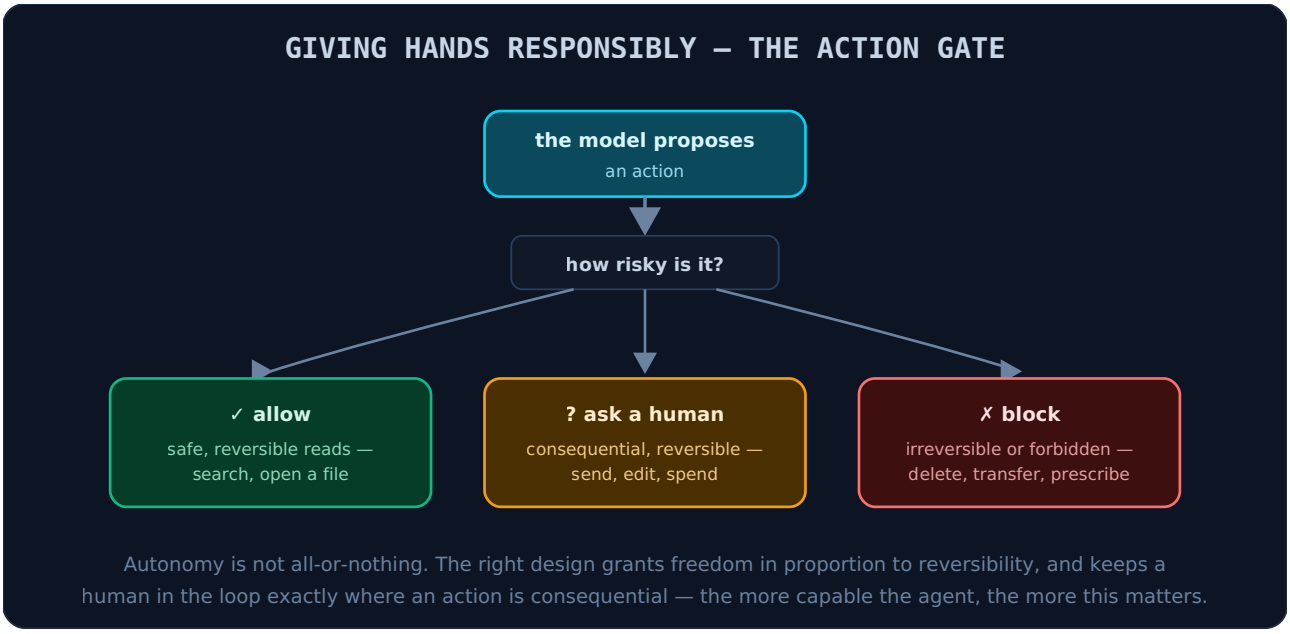

Figure 4. The discipline that makes agency safe: a gate on every proposed action, granting autonomy in proportion to how reversible the action is. Reading is cheap and freely allowed; sending, editing, and spending pause for a human; the irreversible and the forbidden are blocked outright. This is not bureaucratic caution — it is the direct acknowledgement that a system which can be confidently wrong should not also be able, unsupervised, to do the irreversible.

> **The line I would draw in medicine, and why.** An agent with read-only access to the literature and a patient's record is a research assistant — a genuinely useful one. An agent with *write* access to the record, or worse, the ability to order and prescribe, is a categorically different risk, because now a hallucinated fact or a mis-retrieved guideline does not produce a wrong paragraph on a screen — it produces a wrong action in a patient's care. Everything this book taught you about how these systems fail is an argument for keeping the human decisively in the loop at exactly the point where the agent's proposal becomes an act. Understanding the machine is not a reason to trust it more. It is the reason to know precisely where not to.

The whole machine

And so we arrive at the end, with the machine complete. Let us see all of it at once.

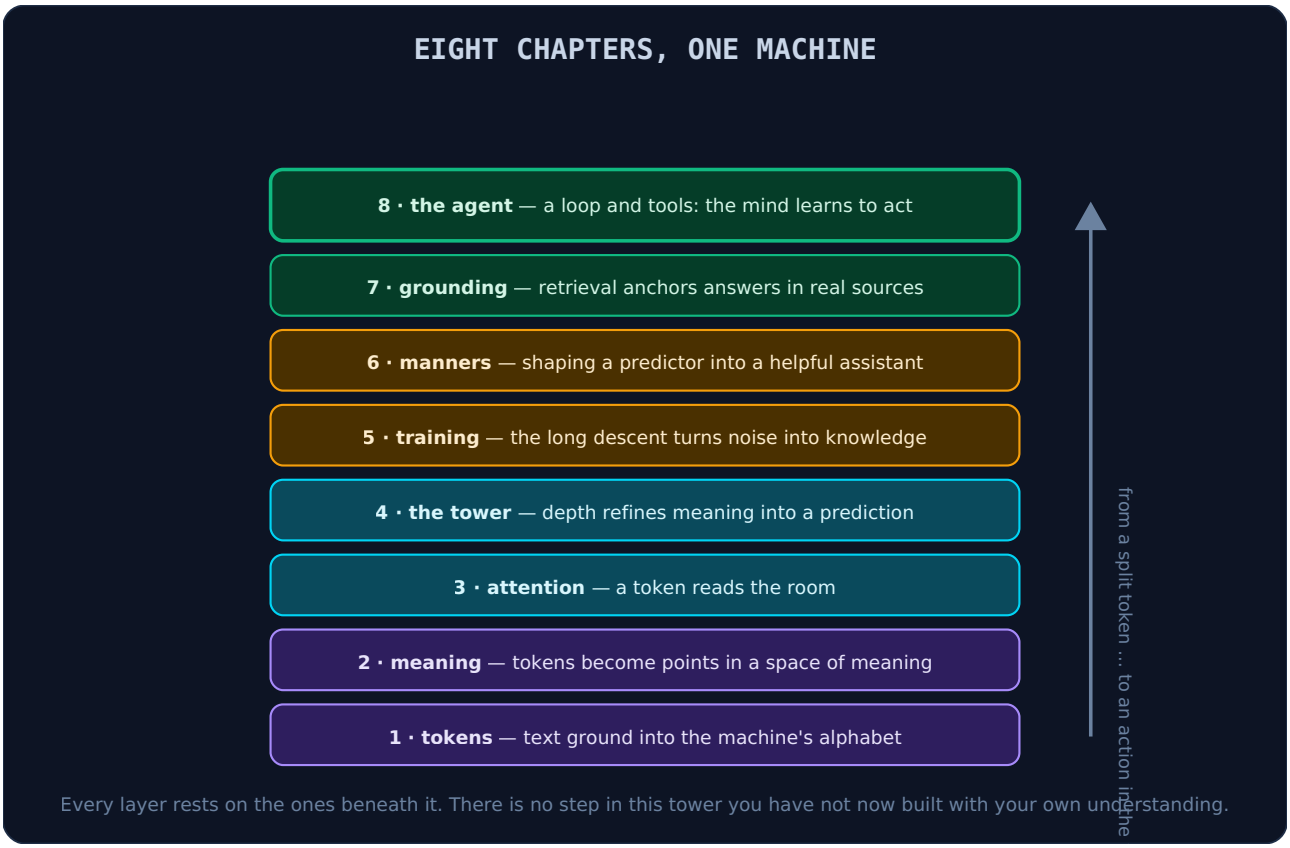

Figure 5. The complete machine, eight chapters in one image. Read it from the bottom up and it is the arc of the whole book: a sentence is split into tokens, the tokens are given meaning, they learn to read each other, they are refined through a deep tower, the tower is filled by training, shaped into an assistant, grounded in real documents, and finally — wrapped in a loop with tools — set free to act.

Return, for a moment, to where we began. Chapter One opened with you typing a sentence and pressing enter, and with the quiet, unremarkable-seeming act of that sentence being taken apart. I said then that everything the machine does happens downstream of that first disassembly, and that if you wanted to understand these systems from the ground up, that was the ground.

You can now follow that sentence the entire distance. You can watch it shatter into tokens, each token find its place in a landscape of meaning, the tokens turn and read one another through attention, the whole rise up a deep tower that was itself filled, weight by weight, by the long patient descent of training. You can see how a mere predictor of text was shaped into something that answers helpfully, then given the means to look things up and show its working, and finally set inside a loop where it can reach out and act. None of it is magic to you any longer. All of it is machinery you understand.

That understanding is not merely satisfying. It is, increasingly, a form of responsibility. These systems are being handed into medicine, into law, into the places where being confidently wrong has a cost measured in human terms. The people who understand

them from the inside — who know not just what they can do but exactly how and where they fail — are the ones who can deploy them with the care the stakes deserve. You are now, whatever your field, one of those people. That was the whole point.

The machine is built. What you choose to build with it is the part no book can write for you.

— *Neal*

↑ Back to Contents