

chapter one

What a Language Model Actually Is

This chapter models text generation, not every computation in a deployed system.

Otto will draft support replies, retrieve documents, and request tools. This chapter isolates one operation used during text generation: given the available context, a model assigns scores to possible next tokens. A decoding procedure selects a token, appends it, and repeats.

That account is useful but deliberately incomplete. Deployed models may accept images or audio, perform hidden or exposed reasoning, and use provider-specific processing around generation. The application still receives an autoregressively generated token sequence, but the visible reply is not necessarily the model's only computation.

The core move: read text, predict the next piece

Give the model an encoded context. During generation it computes scores for possible continuations. Suppose the visible text ends:

Hilltop Coffee ships within 3 to 5 business

days is a plausible high-scoring continuation of this familiar phrase. The example is illustrative; it does not report measured scores from the stand-in model used later in the book.

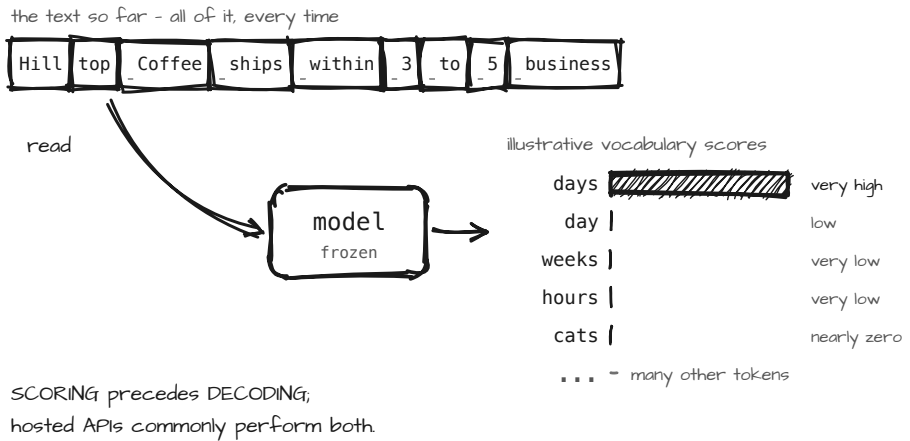
Internally, the generation step produces a score for tokens in the model vocabulary. The provider normally

performs decoding and returns selected output tokens rather than the full score vector. After "3 to 5 business," an illustrative ranking is:

Candidate next token	Score (higher = likelier)
days	very high
day	low
weeks	very low
hours	very low
cats	nearly zero

We'll make these numbers precise in Chapter 4, where you learn how raw scores become probabilities and how you turn them into an actual choice. For now, hold the shape of it: the model's output is a set of next-token scores. A normalization and decoding policy turn those scores into a selected token.

Scoring and decoding are distinct concepts, though an API may perform both.



A conceptual generation step: context produces vocabulary scores, before decoding selects a token. Most hosted APIs do not return this full vector.

note

A token is a chunk of text — often a word, sometimes a word piece or a single character. Chapter 2 is devoted to what tokens actually are; for now, read "token" as "roughly a word."

From one prediction to a whole answer: the loop

One generation step selects one token. To produce a longer reply, the system appends that token and computes the next-token scores conditioned on the extended sequence. This is **autoregressive generation**. Implementations commonly cache intermediate state rather than recomputing the entire prefix from scratch.

Watch it unfold. Otto is drafting a reply and has produced "Your refund" so far:

```
Step 1  input: "Your refund"
        predict: " will"      -> append

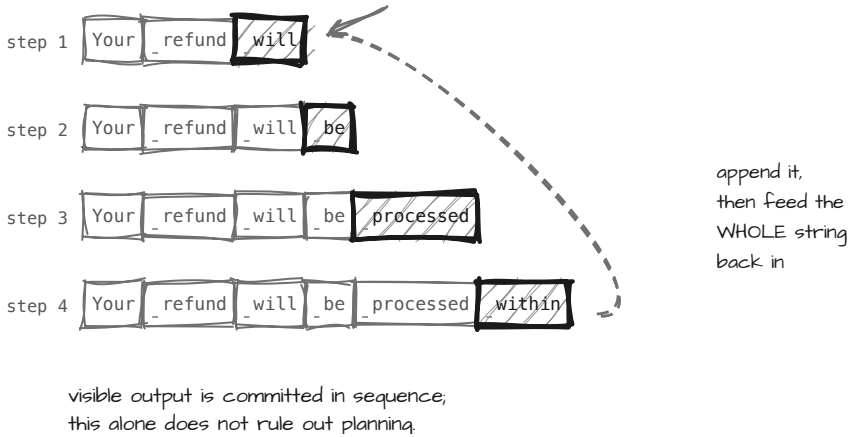
Step 2  input: "Your refund will"
        predict: " be"       -> append

Step 3  input: "Your refund will be"
        predict: " processed" -> append

Step 4  input: "Your refund will be processed"
        predict: " within"   -> append
```

Autoregressive
generation
conditions each new
token on the
preceding sequence.

The visible text grows one token at a time. Generation can stop on a model stop token, a configured stop sequence, a provider rule, or an output-token limit. Inspect the API's stop reason rather than assuming every response ended naturally.



Four conceptual decoding steps. The grey cells precede the newly selected cell; implementations may cache their internal representation.

Each visible output token is emitted in order and is not revised in place by ordinary streaming generation. This does not establish that a current model performs no planning. Some APIs expose or hide reasoning tokens, and an application can request a plan, critique a draft, or use several calls. The narrower engineering fact is that the final visible sequence remains committed token by token unless the application replaces or regenerates it.

Visible tokens are committed in order; planning behavior depends on model and scaffold.

What "learned from text" actually means

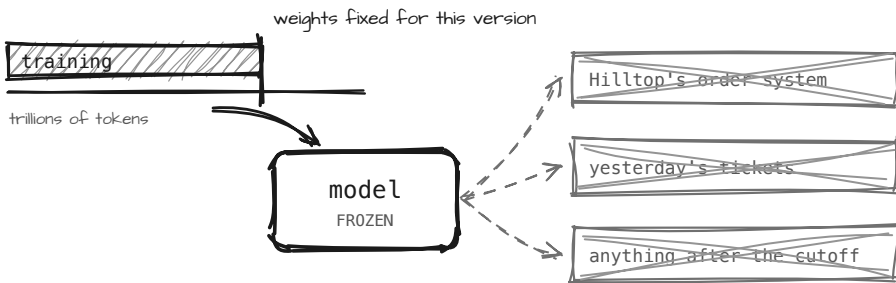
During pretraining, a model adjusts its **parameters** to reduce a prediction loss over large datasets. Deployed chat models usually also undergo post-training intended to shape instruction following, safety behavior, and other response properties. Providers do not necessarily publish complete datasets or training recipes.

Parameters, also called weights, are learned numeric values. The completion in this example can reflect patterns captured during training and information supplied

in the current context. It is not evidence that the model consulted Hilltop's policy database.

Fixed weights do not imply fixed inputs; retrieval and tools can supply current data.

For one named model version, the weights used for an inference call are fixed. Current information can still enter through the request, retrieval, or tool results. The base model has no implicit access to Hilltop's database or yesterday's tickets; the application must supply that data through an authorized interface.



no IMPLICIT live lookup or application memory;
current data requires context, retrieval, or tools.

A base model call has no implicit connection to Hilltop systems. Current data must arrive through the request or an explicit tool.

Calling the machine

In practice you don't run the loop yourself; you call an API and the provider runs it for you. Here is Otto answering a ticket with the Anthropic Python SDK:

```

from anthropic import Anthropic

client = Anthropic()

ticket = "Hi, how long does shipping take?"

resp = client.messages.create(
    model="claude-sonnet-5",
    max_tokens=1024,
    system="You are Otto, Hilltop's support "
        "assistant.",
    messages=[{"role": "user",
                "content": ticket}],
)

print(resp.content[0].text)

```

The service converts the request's system instructions, messages, and provider metadata into the model's input format. That format is not specified as simple string concatenation. `max_tokens` sets an output ceiling; if the response reaches it, the stop reason indicates truncation rather than a natural end. Reasoning-capable models may also account for internal tokens according to model-specific rules.

system is a provider-defined request field with model-specific behavior.

note

`model="claude-sonnet-5"` is a stand-in name used throughout this book. Real model names, along with their prices and limits, change often — treat any specific number here as a snapshot, not a constant.

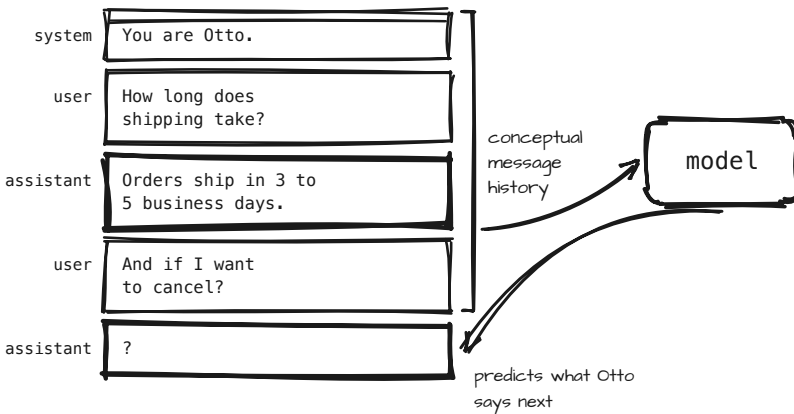
Conversation state

Multi-turn behavior requires earlier turns to remain available to the next request. One conceptual rendering is a role-labelled transcript:

System: You are Otto, Hilltop's support assistant.
 User: How long does shipping take?
 Otto: Orders ship in 3 to 5 business days.
 User: And if I want to cancel?
 Otto: <- model predicts the next token here

Conversation state belongs to the application or service, not an enduring model mind.

The model call can condition on these earlier turns, though the provider controls their internal formatting. Some APIs require the application to resend message history; others accept a conversation or previous-response identifier and manage stored state. Either way, the application must understand retention, storage, context limits, and deletion behavior rather than assuming the model remembers a user independently.



Conversation state belongs to the application or service.
 APIs may resend messages or reference stored state.

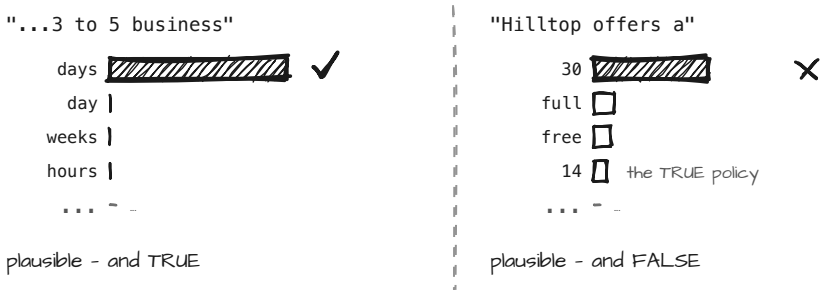
A conceptual message history supplied for the next turn. Exact formatting and whether history is resent or referenced depend on the API.

What this model explains

Next-token training helps explain why fluent form does not certify a claim. Otto can produce a conventional refund email while supplying a policy absent from

Hilltop's records. We cannot infer the exact cause of one token from an undocumented training corpus, and modern post-training complicates the simple “most common phrase” story. The operational lesson is narrower: generation does not perform an authorized Hilltop lookup unless the application provides one. Retrieval can supply evidence, but the system must still validate sources, citations, and the resulting answer.

Fluency is not evidence that a claim came from an authorized source.



Both rankings are illustrative.
Only an authorized source establishes policy.

Illustrative continuations can both be fluent while only one matches Hilltop policy. The bars are not measured model probabilities.

Three points carry forward. Text generation commits visible tokens in sequence. Hosted APIs combine that mechanism with versioned message formats, decoding, and sometimes reasoning features. Fixed weights do not provide implicit access to application data; current facts and conversation history require explicit context, stored service state, retrieval, or authorized tools.

The next chapter examines the tokens counted by those interfaces.

Exercises

Exercise 1.1. Otto is drafting a reply and has produced the text "Your refund will be" so far. Using the autoreg-

ressive loop from this chapter, write out the next two steps of generation the same way the chapter does (input, predicted token, append). Name the single operation the model performs at each step.

Exercise 1.2. Explain the difference between committing visible output tokens autoregressively and claiming that a model performs no planning. Name two ways a model or application can plan or revise, and state why an already streamed token still requires replacement or re-generation to change.

Exercise 1.3. A colleague says, "Otto remembers our chat." List two API designs that can make earlier turns available: resending history and referencing provider-stored state. What retention, deletion, and context-limit questions must the application answer?

Exercise 1.4. Otto confidently tells a customer that Hilltop offers a "30-day money-back guarantee," which Hilltop never wrote. Explain why fluent generation is insufficient evidence for this claim. Name the authorized data source and validation step Otto would need before including the policy in a reply.