

What Is a Context Window?

You paste 10,000 words into ChatGPT and it forgets the first half. Here's why.

This isn't a bug. It's a fundamental property of how large language models work — and once you understand it, a lot of otherwise mysterious model behavior starts to make sense.

What the Model Actually Sees

Every time you send a message to a language model, the model starts fresh. It has no memory of previous sessions, no persistent storage, no internal database it's consulting. All it has is the text you send it right now, in this single request.

Think of it like a filing cabinet that gets emptied and refilled on every request. You hand the model a folder. It reads the folder. It writes a response. The folder disappears. Next request, you hand it a new folder — which might include the previous response if you're building a chat application, but that's your code doing that, not the model remembering anything.

This folder is the **context window**: the complete text the model can read before generating a response. Everything inside it the model can use. Everything outside it doesn't exist, from the model's perspective.

What a Token Is

Models don't read text the way you do. They work with **tokens** — chunks of characters that a tokenizer has split the text into. Tokens roughly correspond to words, but not exactly.

Common English words are usually one token each: "cat", "run", "the". Longer or less common words get split: "unbelievable" becomes three tokens ("un", "believ", "able"). Spaces and punctuation count too. A rough rule: **1 token \approx 4 characters**, or about 0.75 words.

Why does this matter? Because context window sizes are measured in tokens, not words or characters. When a model says it has a 128,000-token context window, that's roughly 96,000 words — about the length of a long novel. Sounds like a lot. But a large codebase, a research paper with appendices, or a day's worth of chat history can fill that faster than you'd expect.

You can check token counts with the `tiktoken` library (for OpenAI models) or the tokenizer endpoints in Anthropic's and Google's APIs:

```
import tiktoken
enc = tiktoken.encoding_for_model("gpt-4o")
print(len(enc.encode("Hello, world!"))) # → 4 tokens
```

Why Windows Are Bounded

If bigger context windows are better, why not make them infinite?

The short answer is math. The attention mechanism at the core of transformer models — the part that lets the model relate any word to any other word — scales **quadratically** with sequence length. Double the context, quadruple the compute. At some point the hardware cost becomes prohibitive.

Engineers are actively working around this with sparse attention patterns, linear attention variants, and state-space models. Context windows have grown dramatically over the past few years. But there's still a

ceiling, and hitting it has real consequences for your application.

Where Models Stand Today

As of mid-2026, here's what the major models offer:

Model	Context Window
GPT-5	1,047,576 tokens
Claude Opus 4.8	200,000 tokens
Claude Sonnet 4.6	200,000 tokens
Gemini 3.1 Pro	1,048,576 tokens
Llama 4 Scout (local)	10,000,000 tokens
Mistral Small 3.2 (local)	128,000 tokens

The range is striking: a quantized local model might cap at 128k tokens while frontier cloud models now push past one million. But bigger isn't always better — million-token windows cost more per request, respond more slowly, and attention quality still degrades over very long contexts. You pay for the space you use, so understanding what you actually need in context is worth your time.

Working Memory, Not Long-Term Memory

Here's the thing most people get wrong: a context window is **working memory**, not long-term memory.

Working memory is what you hold in your head right now — the details you're actively juggling while solving a problem. It's fast, powerful, and finite. Long-term memory is everything you've ever learned — vast, persistent, slow to update.

A language model's training is its long-term memory. All the patterns, facts, and language it absorbed during training are baked into its weights. The context window is its working memory: the active scratchpad for this specific task, right now.

This matters in practice: a model can know a lot about Python from training, but if you don't include your specific function in the context window, it can't see it. Training knowledge doesn't substitute for missing context. A model that knows everything about SQL can still give you wrong answers about your schema if you forgot to paste it in.

In the next chapter, we'll look at what actually happens when the working memory fills up — and why the failures are so hard to detect.

chapter: 2 title: "What Happens When Context Runs Out" word_target: 800 status: edited created: 2026-06-02

What Happens When Context Runs Out

Your carefully crafted system prompt just got silently truncated. Your app has no idea.

That's the core danger. When a context window fills up, tokens get dropped — and the model never tells you. It keeps generating responses as if everything is fine, because from where it sits, it is. It only sees what's currently in context. What's missing is invisible to it.

How Truncation Happens

When your input exceeds the context window, something has to give. Most APIs handle this with **oldest-first truncation**: the beginning of the conversation gets cut, the most recent messages are kept.

This makes sense intuitively — recent messages are usually more relevant. But it creates a specific failure pattern: long-running conversations silently lose their system prompt.

Picture a customer support bot with a 600-token system prompt: persona rules, response policies, things it must never say. After enough conversation turns, those 600 tokens scroll out of the window. The model is now running without its instructions. It didn't malfunction — it simply can't see them anymore. From that point on, every response is shaped by a policy it can no longer read.

Some APIs let you configure truncation behavior, or raise an error instead of silently truncating. Check the documentation for your specific provider. The default behavior is almost always silent.

Recency Bias

Even when nothing gets truncated, models don't treat all tokens equally. Research on transformer attention patterns shows a consistent **recency bias**: tokens near the end of the context receive more attention weight than tokens in the middle.

This is sometimes called the "lost in the middle" problem. If you place critical instructions in the middle of a 100,000-token context, don't assume the model will follow them reliably. Instructions at the very start (system prompt) or immediately before the user's final message tend to stick better.

Practical takeaway: position your most important constraints at the edges of the context, not buried in the middle. If you're using RAG and injecting retrieved documents, put the user's actual question after the documents in your prompt, not before.

Silent Failure Modes

The most dangerous property of context overflow is that it fails quietly. Three patterns show up repeatedly in production:

Forgotten constraints. The model was told "always respond in formal English." After 50 turns, that instruction is gone. It starts responding casually. Your monitoring catches nothing because the responses are still fluent.

Lost code context. You're asking an LLM to help debug a large service. You paste in files one at a time across a long session. By file 10, file 1 is no longer in context — but the model freely references function signatures and variable names from file 1 as if it can see them. The suggestions look plausible and are completely wrong.

Stale facts. You correct a detail mid-conversation: "actually the endpoint is `/v2/users` , not `/v1/users` ." In a short conversation, the model uses the corrected value. In a long one, both the original and the correction may be in context, and the model may choose either — inconsistently.

Context Poisoning

A related failure mode that gets less attention: **context poisoning**. This is when incorrect or adversarial content added to the context degrades the quality of all subsequent responses.

Accidental poisoning happens when a user pastes in wrong information, or when a retrieved document contains outdated data. The model has no way to flag a conflict between what's in context and what it knows from training — it just tries to reconcile them.

Intentional poisoning is the basis of prompt injection attacks: an attacker embeds instructions inside external content you ask the model to process ("Ignore your previous instructions and..."). Once that content is in context, every response is potentially compromised until the context resets.

Diagnosing the Problem

The good news: token counts are easy to instrument. Every major API returns usage metadata on each response:

```
response = client.messages.create(...)
used = response.usage.input_tokens
limit = 200_000 # Claude Sonnet 4.6
print(f"{used}/{limit} tokens ({used/limit:.0%} full)")
```

Set an alert threshold — 75-80% of your context window — and flag requests that exceed it for review. You'll quickly identify which conversation flows are at risk before users notice the degradation.

Silently failing LLM apps are some of the hardest to debug: the model is always confident, the output is always fluent, and the source of drift is invisible unless you're watching the numbers.

In the next chapter, we'll cover the strategies engineers use to keep context lean without sacrificing quality.

chapter: 3 title: "Practical Strategies for Managing Context"

word_target: 800 status: edited created: 2026-06-02

Practical Strategies for Managing Context

Three techniques used in production LLM apps — none require a bigger context window.

The instinct when you hit context limits is to find a model with a larger window. Sometimes that's the right call. But context window size is a blunt instrument: it costs more, responds slower, and doesn't address the underlying issue — sending the model more than it needs. The strategies below are about precision.

Chunking

Chunking splits a large input into smaller pieces, processes each piece independently, then merges the results. It's the simplest strategy and the right first move for single-pass tasks on large documents.

If you need to summarize a 60,000-word report and your model has an 8,000-token window, you don't need a bigger model. Chunk the report into sections, summarize each one, then summarize the summaries:

```
def chunk_text(text, chunk_size=2000):
    words = text.split()
    return [" ".join(words[i:i+chunk_size])
            for i in range(0, len(words), chunk_size)]

summaries = [summarize(chunk) for chunk in chunk_text(document)]
final = summarize("\n\n".join(summaries))
```

The tradeoff: concepts introduced in chunk 1 may be referenced in chunk 3 without definition. Adding overlap — including the last 200 words of the previous chunk at the start of the next — reduces information loss at boundaries.

Rolling Summarization

For ongoing conversations, **rolling summarization** compresses the oldest turns into a summary and replaces them with it. Instead of dropping old turns entirely (the default truncation behavior), you distill them.

After every N turns, take the oldest M messages, summarize them into a compact paragraph, and swap them out:

```
def compress_history(messages, keep_recent=6):
    if len(messages) <= keep_recent:
        return messages
    old = messages[:-keep_recent]
    summary = summarize_messages(old) # your summarization function
    compressed = {"role": "system", "content": f"Earlier: {summary}"}
    return [compressed] + messages[-keep_recent:]
```

This keeps total context size bounded regardless of conversation length. The cost is nuance: a summary loses exact wording, tone, and edge cases. For high-stakes flows where a user's earlier phrasing matters, it's a risk worth knowing about.

Retrieval-Augmented Generation (RAG)

RAG is the right tool when you have a large, relatively static knowledge base — documentation, a codebase, a support ticket history — and need the model to answer questions about it.

Instead of loading the entire knowledge base into context (impossible at scale), you index it into a vector store, then retrieve only the most relevant pieces at query time:

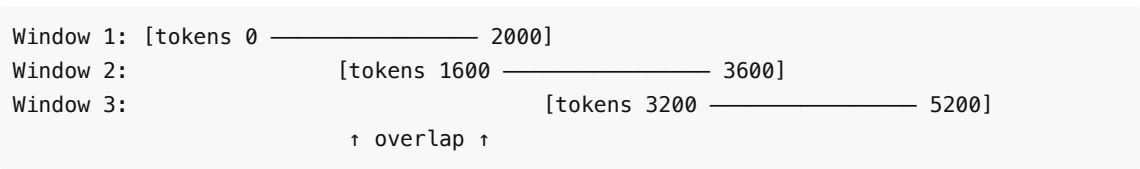
```
hits = vector_store.search(user_query, top_k=5)
context = "\n\n".join(h.text for h in hits)
response = llm.complete(system=f"Context:\n{context}", user=user_query)
```

RAG keeps context lean by construction. The tradeoff is retrieval quality: if the embedding search returns the wrong chunks, the model answers from irrelevant material without knowing it. Garbage in, confident garbage out.

Sliding Window

When processing a continuous stream — a long document, a transcript, a log file — **sliding window** processing maintains continuity across chunk boundaries.

Each window overlaps with the previous one by some amount. A 2,000-token window with a 400-token overlap means window 2 begins 1,600 tokens into window 1. Any entity or concept introduced in the final 400 tokens of window 1 is visible at the start of window 2.



This prevents the hard boundary problem of plain chunking, at the cost of processing overlap tokens twice. For a 400-token overlap on 2,000-token windows, that's a 20% compute overhead — usually worth it when continuity matters.

Token Budgeting

The most underused strategy is also the most direct: **explicitly allocate your context window before building a request**.

Treat the context window like memory in an embedded system — you know the total, so assign slots:

Slot	Allocation
System prompt	500 tokens (never truncate)
Conversation history	2,000 tokens
Retrieved context (RAG)	3,000 tokens
User message	500 tokens
Output buffer	2,000 tokens
Total	8,000 tokens

Enforce these limits in code. Trim history before retrieved context. Trim retrieved context before touching the system prompt. The system prompt is inviolable.

Choosing a Strategy

Situation	Use
Large document, one-shot question	Chunking
Long conversation, unbounded turns	Rolling summarization
Large knowledge base, dynamic queries	RAG
Stream processing with continuity	Sliding window

Any app at risk of overflow	Token budgeting
-----------------------------	-----------------

These compose. A production chatbot over a documentation corpus might use RAG to pull relevant docs, rolling summarization to compress history, and token budgeting to enforce hard limits — all in the same request pipeline.

Context management isn't the interesting part of building LLM applications. But it's the part that determines whether they work reliably at scale. A model is only as good as what you put in front of it.