Day 1: Introduction to LLM Context Engineering

Mental Model & The Six Core Elements

Isham Rashik · AI Engineer
Engineering AI with Clarity
NLP • Computer Vision • Fine-Tuning • RAG • Agentic AI

Version: v2.0 · April 25, 2026

Acknowledgment

I am deeply grateful to Dr. Sreedath Panat and **Vizuara Technologies** for creating and generously sharing the *Context Engineering* course. These notes would not exist without his exceptional teaching, clear explanations, and dedication to making cutting-edge AI concepts accessible to everyone.

Thank you, Dr. Panat, for the time, effort, and passion you have poured into this course. Your work has been instrumental in shaping my understanding of context engineering.

→ Watch the full playlist on YouTube

Contents

1 Overview

This guide introduces the foundational mental model for **LLM Context Engineering**, the discipline of designing, assembling, and managing everything that flows into a large language model's context window to maximize output quality. It establishes why context engineering has become the defining skill for AI practitioners in 2025–2026, contrasts it with traditional prompt engineering, introduces the **six core elements** of the context window, and demonstrates how each layer progressively improves LLM output quality.

To understand why this discipline matters, we begin with a real-world problem that motivates the entire discussion.

2 Why Context Engineering? The Motivation

The explosion of coding agents (Claude Code, Codex, Cursor, Copilot, OpenClaw) in 2025–2026 has revealed that **the quality of an LLM's output is primarily determined by the quality and structure of its context**, not just the user's prompt. Context engineering is the skill that separates tools that work "80% of the time" from tools that produce reliably excellent output.

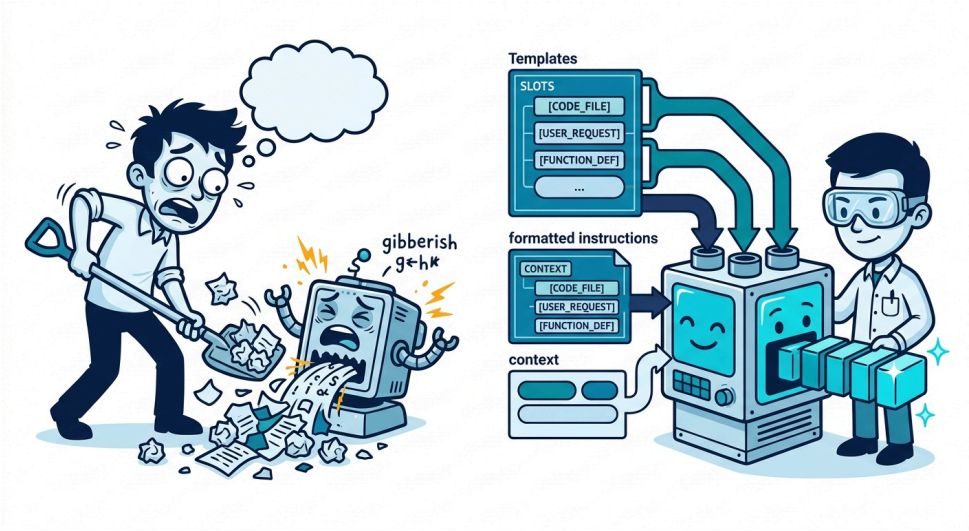

Figure 1: Left: "Just wing it." Right: "We engineered the context." Guess which one got promoted.

2.1 The Email Agent Case Study

Consider a concrete problem: building an AI agent to process and respond to ~200+ emails per day, replacing 2.5–3 hours of daily manual work. The key insight is that a simple chatbot with

a user prompt is woefully inadequate. The agent needs **rich, structured, multi-dimensional context** to produce responses that match a human expert's quality.

2.2 Elements Needed for a Real-World Agent

From the email agent example, the following context requirements emerged:

Table 1: Context requirements for a real-world email agent

Element	Why It's Needed	Challenge
Website metadata	Answer questions about courses, programs, policies	100+ courses cannot all fit in context; need compressed metadata
Email history	Learn the user's writing style, tone, preferences	10,000+ threads = ~250 MB of JSON; far too large for direct context
Recent emails (recency bias)	Last 20-30 emails contain the most current, accurate information	Must be prioritized over older, potentially stale data
Rules / Guardrails	Global policies (e.g., "always share course-name links, not ID links")	~10 rules that must never be violated
Feedback system	User rates responses (1-5), edits drafts - agent learns over time	Feedback grows unboundedly; cannot load 10,000 feedback items
RAG retrieval	Fetch top-K relevant course details for a specific query	Must be low-latency, high-relevance
Incoming email (user input)	The actual message to respond to	Just one of many context elements

> **Key Insight**
>
> The user's input (the incoming email) is a tiny fraction of the total context. Most of the context is engineering (retrieval, memory, system instructions, tool outputs), all of which must be carefully designed.

These requirements make it clear that a single prompt is no longer enough. But where does context engineering sit among the broader set of approaches practitioners use to interact with LLMs?

3 The Engineering Spectrum

The way practitioners interact with LLMs has evolved rapidly: from hand-crafting single prompts, to iterating by feel, to engineering entire context pipelines, to orchestrating autonomous agents. Each approach builds on the last, and understanding where each sits on this spectrum is essential for choosing the right tool for the job.

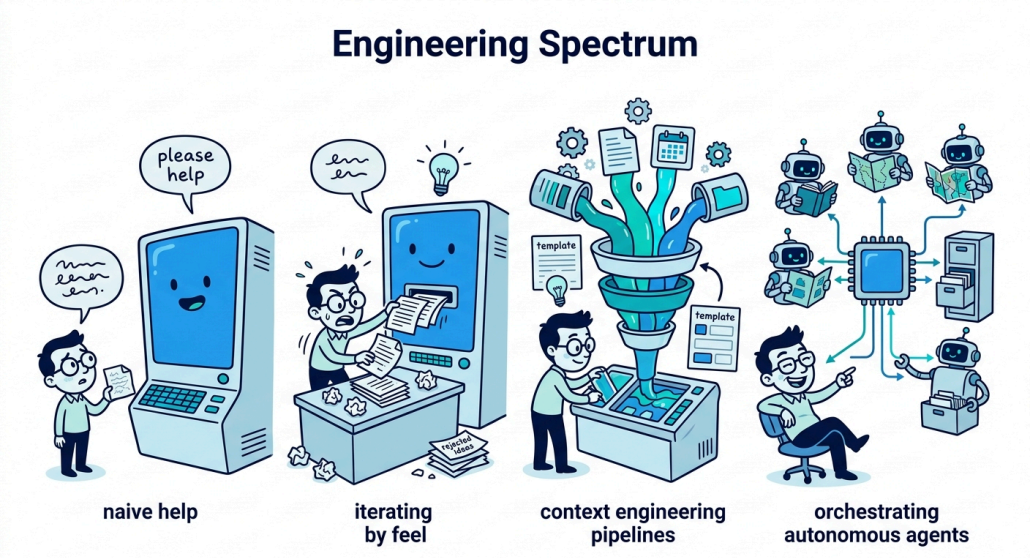

Figure 2: Evolution of an AI practitioner: "please help" → "here's your entire life story, now help"

3.1 Prompt Engineering: Single-Dimensional Thinking

Prompt engineering is the practice of crafting a **single text input** to an LLM to maximize output quality. The user goes to a chat interface (e.g., chat.ai.com), types a request, and iterates. Key characteristics:

- Input is a **text string**, the user's natural language query
- Goal: craft text so the LLM understands your **intent**
- Only **one API call** at a time (user asks, model responds)
- Techniques: one-shot examples, few-shot examples, specifying persona, controlling output format
- Good for beginners, but fundamentally limited

3.2 Context Engineering: Multi-Dimensional Thinking

Context engineering encompasses **everything** that fills the LLM's context window: not just the user prompt, but system instructions, retrieved documents, tool outputs, conversation history, memory, and state. Key characteristics:

- Input is a **composite assembly** of multiple information sources
- Involves designing **dynamic systems** that assemble the right context at inference time
- Multiple API calls, tool invocations, retrieval pipelines run in concert
- Requires engineering of RAG, MCP, memory systems, guardrails
- The user prompt is just one of six (or more) elements

3.3 Agent Engineering: Autonomous Task Execution

Agent engineering is the practice of defining agents, their tools, skills, and output formats so they can execute tasks autonomously. Key characteristics:

- **Agent engineering:** Define agents, their tools, skills, and output formats. Agents can execute tasks autonomously.
- **Context engineering:** Manage what information each agent (or the overall system) receives in its context window. This is required *regardless* of whether you use agents or not.
- Key distinction: When you have **multiple agents** (sub-agents), each agent gets its **own context window**. Context engineering determines what goes into each agent's context independently.

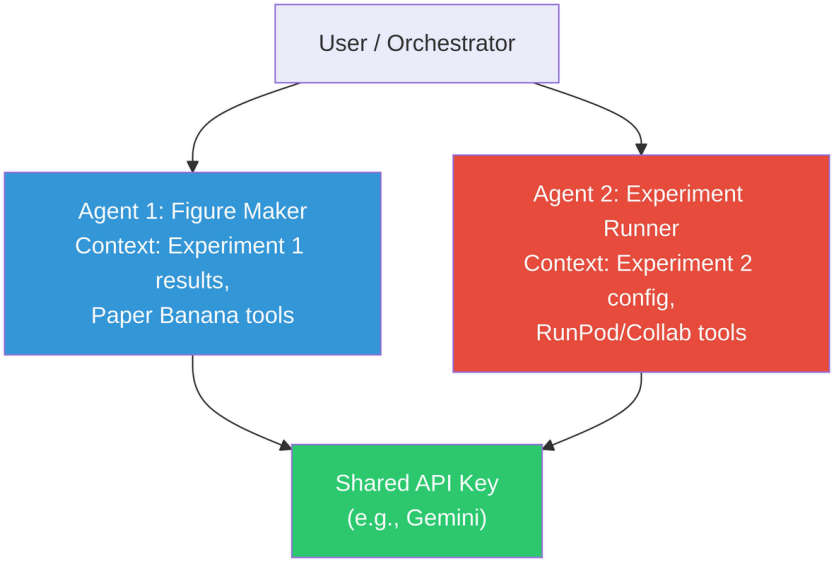

Figure 3: Sub-agent context isolation: each agent gets its own context window

Walkthrough:

1. Two agents share the same API key but have **separate context windows**.
2. Agent 1 (figure-making) only needs Experiment 1 results and Paper Banana tools in its context.
3. Agent 2 (experiment runner) only needs Experiment 2 configuration and compute tools.
4. Loading Experiment 2 details into Agent 1's context would be wasteful and could cause context rot.
5. Each agent's context is independently engineered for its specific task.

> 🔥 **Claude Code**
>
> Sub-agents automatically get their own context window, so whatever is happening in one session doesn't pollute another's context. However, one sub-agent may produce poor results if *its* context exceeds the optimal limit, even while another sub-agent performs well.

3.4 Vibe Coding: Iterative, Feeling-Based Interaction

Vibe coding (term coined by Andrej Karpathy) refers to iteratively giving prompts to an LLM and adjusting based on output, essentially "coding by feel." The user is not writing code explicitly but sharing instructions and reacting to what the LLM produces.

Context engineering differs in that while the *user input* portion might involve some "vibing," the other five elements (RAG, tools, MCP, memory, system prompt) are **pure engineering**. They require deliberate, precise design:

- RAG system must have low latency and high relevance metrics
- MCP must provide access to specific backend systems
- Memory must be explicitly partitioned into persistent and short-term
- System prompts must be carefully crafted (cannot be "vibed")
- Markdown files (CLAUDE.md) must be explicitly engineered

3.5 Comparison Table

Table 2: Prompt Engineering vs. Vibe Coding vs. Context Engineering vs. Agent Engineering

Aspect	Prompt Engineering	Vibe Coding	Context Engineering	Agent Engineering
Input	Single text string	Iterative prompts adjusted by feel	Composite multi-source assembly	Task definition + tool schemas
User's role	Craft the perfect prompt	React to LLM output, adjust on the fly	Design the system that assembles context	Define agents, tools, skills, and orchestration
Dimensions	One (the prompt)	One (the prompt), iterated	Six+ (system prompt, RAG, tools, memory, history, user input)	Agents + their individual context windows
API interaction	One call at a time	Multiple calls, trial and error	Multiple calls, tools, retrieval pipelines	Autonomous multi-step execution with tool use
Dynamic content	No (static prompt)	No (user adapts manually)	Yes (RAG, tool outputs, live memory)	Yes (agent decides what to retrieve and when)
Skill level	Beginner-friendly	Beginner-friendly	Requires engineering expertise	Requires systems architecture expertise
Buzzword era	2023	2024-2025	2025-2026	2025-2026

With this spectrum in mind, we need a mental model for how all these pieces fit together inside the LLM ecosystem. Andrej Karpathy offers a compelling one.

4 The LLM OS Analogy (Andrej Karpathy)

Andrej Karpathy delivered a widely-viewed talk at Y Combinator (late 2025) [1] where he drew an analogy between an operating system and the LLM ecosystem. This analogy provides a powerful mental model for understanding the role of each component.

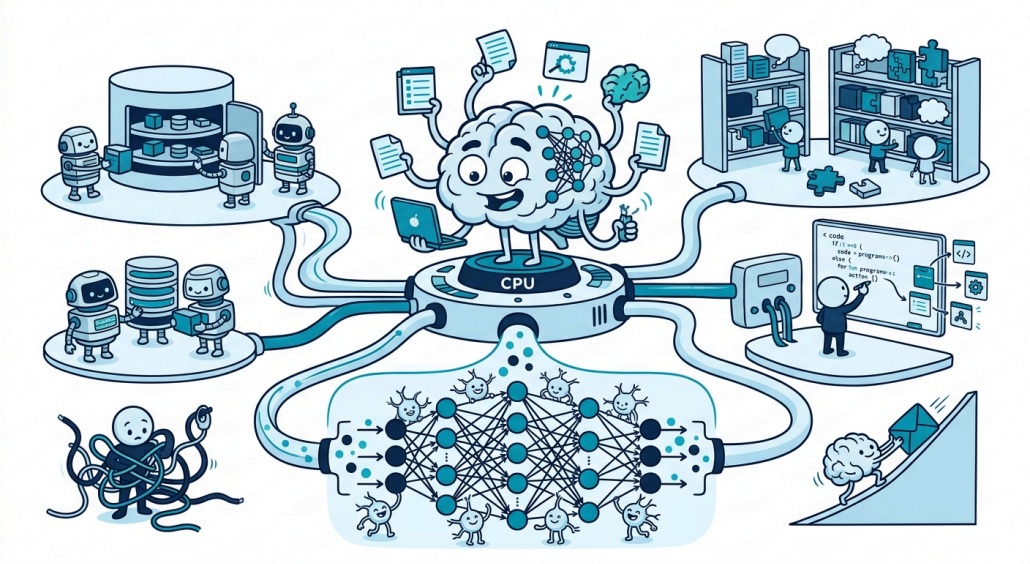

Figure 4: Your LLM is basically Windows Vista but with better autocomplete

4.1 Component Mapping Diagram

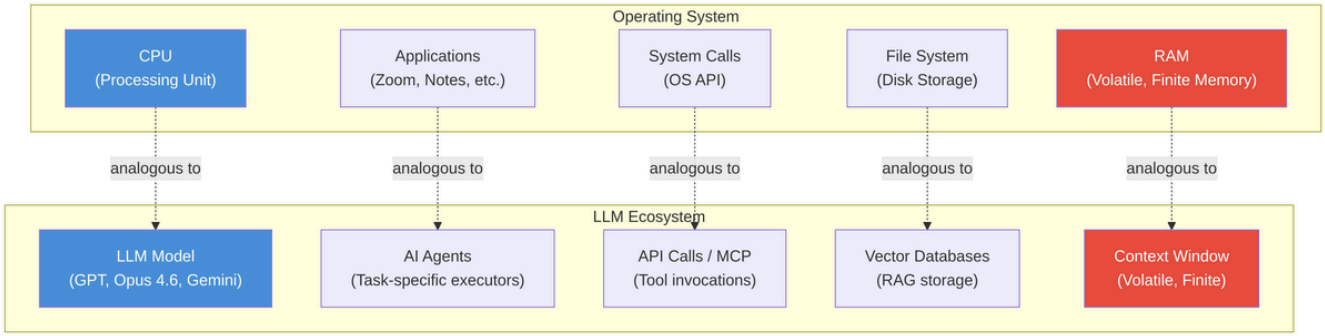

Figure 5: OS to LLM ecosystem component mapping

Walkthrough:

1. The **CPU** is the processing engine of an OS. Similarly, the **LLM model** (GPT, Opus 4.6, Gemini) is the processing engine of an AI system.
2. **Applications** (Zoom, notes apps, etc.) are analogous to **AI agents**, each performing a specific task.

3. **System calls** that applications use to interact with the OS are analogous to **API calls / MCP**, the standardized interface through which agents interact with tools and external services.
4. The **file system** (disk storage loaded into RAM on demand) is analogous to **vector databases** used in RAG: persistent storage that is selectively loaded into context.
5. **RAM** is analogous to the **context window**. Both are volatile (not permanent), finite (limited capacity), and serve as the working memory for active processing.

4.2 Component Mapping Table

Table 3: OS to LLM ecosystem mapping

OS Component	LLM Equivalent	Key Property
CPU	LLM model (GPT, Opus, Gemini)	Core processing unit
Applications	AI Agents	Task-specific executors
System Calls	API calls / MCP (Model Context Protocol)	Standardized tool interaction
File System	Vector Databases (RAG storage)	Persistent, on-demand access
RAM	Context Window	Volatile, finite, working memory

> **? Why is context window = RAM?**
>
> Two constraints are shared: (1) RAM is not permanent memory; context window contents change with each interaction. (2) RAM is not unlimited; even a 2M token window is finite and must be managed carefully.

The RAM analogy naturally raises a question: how large is this "RAM," and how should we think about its capacity? That requires understanding tokens.

5 Context Window: The RAM of an LLM

The context window is the working memory of an LLM, everything the model can "see" during a single inference call. Understanding its size, unit of measurement, and practical limits is foundational to every context engineering decision that follows.

Figure 6: The context window: like RAM, except when it's full, the LLM doesn't crash — it just gets creative

5.1 Token-to-Word-to-Page Conversion

The fundamental unit of LLM input/output is the **token**. Understanding the relationship between tokens, words, and pages is essential for context budgeting.

Key conversion formula:

$$1 \text{ token} \approx \frac{3}{4} \text{ of a word} \approx 0.75 \text{ words} \tag{1}$$

Table 4: Token-to-word-to-page conversion

Symbol	Meaning	Value
1 token	Basic unit of LLM processing	~0.75 words (English, BPE tokenizer)
1 page	Standard page of a book	~300 words

Derivation for a 2 million token context window:

1. $2,000,000 \text{ tokens} \times 0.75 \frac{\text{words}}{\text{token}} = 1,500,000 \text{ words}$
2. $1,500,000 \frac{\text{words}}{300 \frac{\text{words}}{\text{page}}} = 5,000 \text{ pages}$

Result: A 2 million token context window can fit approximately **5,000 pages** of text.

5.2 Context Window Sizes of Major LLMs

Table 5: Context window sizes of major LLMs

Model	Context Window (tokens)	Approximate Pages
GPT-4o	~128K	~300 pages
GPT-5	Larger (unspecified)	---
Claude Opus	~200K	~500 pages
Gemini 2.0 Pro	~2M	~5,000 pages
Meta Llama 4 Scout	~10M	~28,000 pages

5.3 Why Context Engineering Matters Despite Huge Windows

A natural question: if the context window can hold 5,000+ pages, why bother engineering it?

Three critical reasons:

1. **Cost:** LLM usage costs are proportional to token consumption. More tokens = higher cost per API call.
2. **Latency:** More tokens to process = slower response times.
3. **Context Rot:** Output quality **degrades** as context size increases, even before the window is full. Irrelevant tokens dilute signal-to-noise ratio.

> **Memorize**
>
> Saying "we don't need context engineering because the window is huge" is like saying "we don't need database management because we have 2 TB of RAM." The size of the resource makes management *more* important, not less, because increasingly complex tasks demand it.

So what happens when context is managed poorly? The answer is context rot, one of the most insidious failure modes in LLM applications.

6 Context Rot

Even when a context window is far from full, the quality of the LLM's output can silently degrade. This phenomenon, known as context rot, is the central failure mode that context engineering aims to prevent.

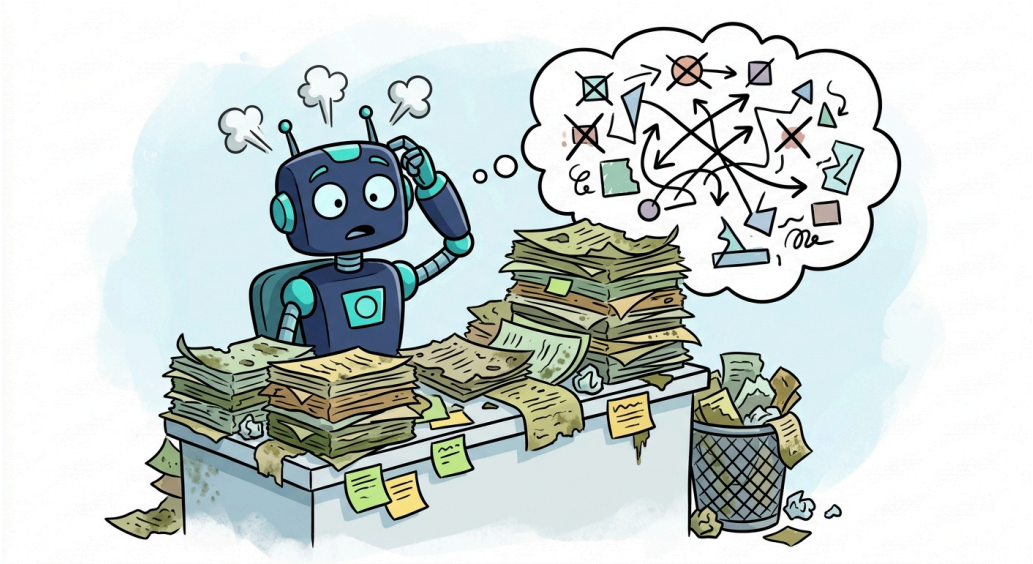

Figure 7: Context rot: your LLM slowly forgetting what you asked while remembering every irrelevant detail

6.1 What Is Context Rot?

Context rot is the **degradation of LLM output quality** caused by excessive, irrelevant, contradictory, or stale information in the context window. It occurs even when the context window is not full.

> **Quote**
>
> Find the smallest set of high-signal tokens that maximize the likelihood of your desired outcome.
>
> — Anthropic

Chroma published a well-regarded article [2] with quantitative analysis showing how context rot intensifies as token count increases within the context window.

6.2 Causes of Context Rot

Table 6: Causes of context rot

Cause	Example	Impact
Stale information	Policy from last week that has since changed	LLM uses outdated facts
Contradictory instructions	"Use font size 50" in one section, "use font size 20" in another	LLM produces inconsistent output
Irrelevant context	Loading all 100 courses when the query is about one	Signal-to-noise ratio drops
Unbounded accumulation	15+ turns of conversation diluting system prompt	System prompt "voice" degrades over time

> ⚠️ **Common Misconception**
>
> **Not a cause of context rot:** Token fragmentation from mixed encoding schemas. This is a tokenization concern, not a context quality issue.

6.3 The Sweet Spot: Balancing Context Size and Quality

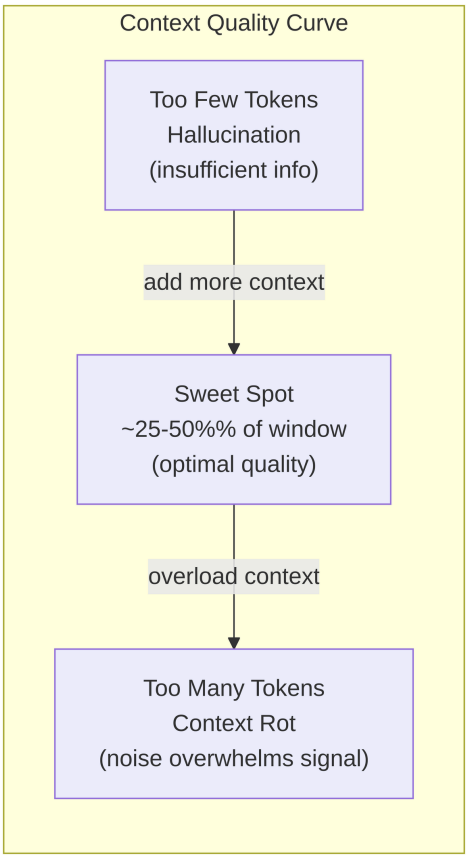

Figure 8: Context quality curve: balancing token count and output quality

Walkthrough:

- **Too few tokens:** The model lacks sufficient instructions and information, leading to hallucination and generic responses.
- **Sweet spot:** For a 200K token window, approximately 50K–100K tokens of context (25–50% of the window) produces optimal quality. The remaining capacity is reserved for model generation output.
- **Too many tokens:** Noise overwhelms signal, relevant information gets buried, contradictions emerge, and output quality degrades.

> 🔥 **Tip**
>
> This is an **empirical** observation, not derived from physics or theory. The exact sweet spot varies by model, task, and content composition.

Even within the sweet spot, not all positions in the context are equal. A landmark research finding reveals exactly where information gets lost.

7 Lost in the Middle Effect

Not all token positions in the context window receive equal attention from the model. A landmark 2023 paper by Liu et al. [3] uncovered a systematic bias in how LLMs process long contexts, with direct implications for how practitioners should order information.

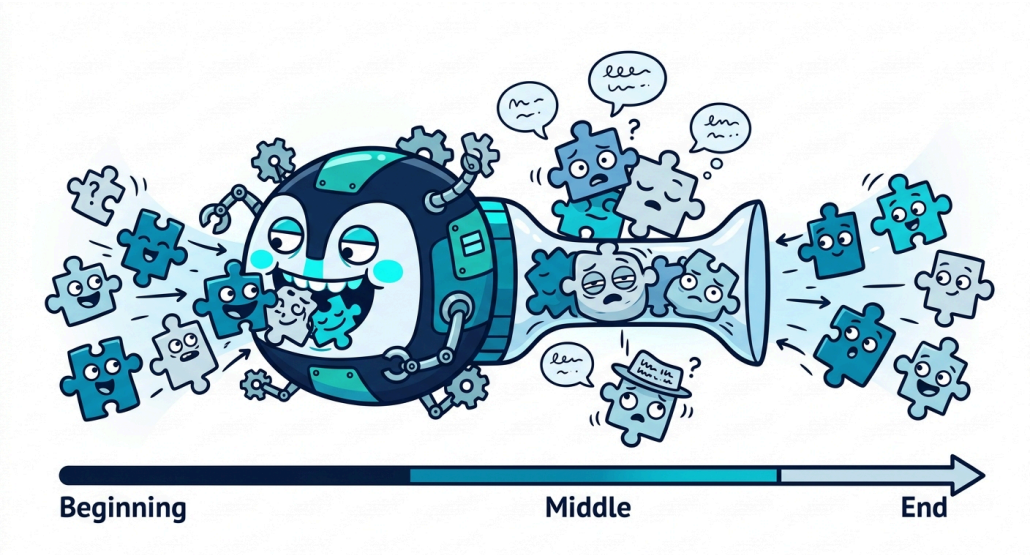

Figure 9: LLMs reading your context: first page? Memorized. Last page? Got it. Pages 3–47? Never heard of them.

7.1 The Core Finding

The paper *"Lost in the Middle: How Language Models Use Long Contexts"* [3] (~3,500 citations) demonstrated that LLMs pay **disproportionately more attention to information at the beginning and end** of the context window. Information placed in the middle of the context is more likely to be overlooked or under-weighted during generation.

Key findings:

- The effect is **more pronounced with larger context windows** (>100K tokens)
- It is an empirical observation with no definitive theoretical explanation, despite the fact that attention mechanisms are designed for long-range dependencies
- It applies to all major LLMs tested

Related Test

The related "Needle in a Haystack" test evaluates whether a model can attend to a single critical token placed at arbitrary positions within a large context, complementary to the "lost in the middle" finding.

7.2 Practical Implications for Context Assembly

The lost in the middle effect directly dictates **how to order elements within the assembled context:**

- **Critical, non-negotiable information** (system prompt, guardrails) ⟶ place at the **beginning**
- **User's current query** ⟶ place at the **end**
- **Dynamic, less-critical content** (RAG chunks, tool outputs) ⟶ fills the **middle**

The goal: **shrink the middle** so that even if some information there gets lost, nothing critical is affected.

7.3 Context Assembly Order Diagram

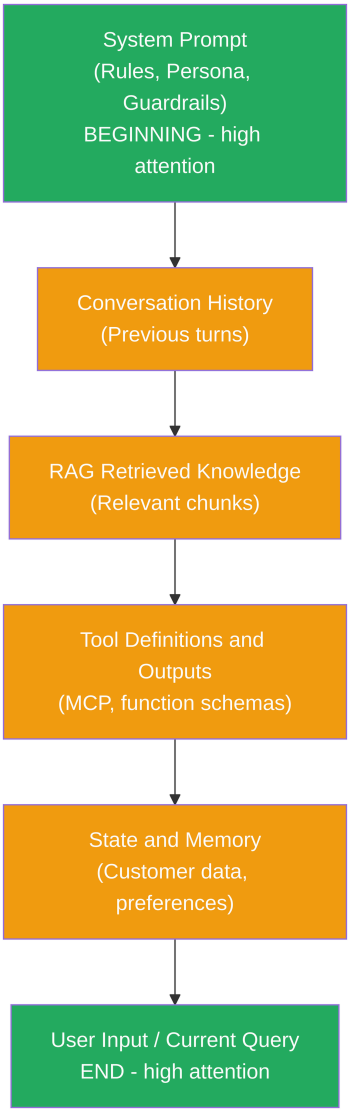

Figure 10: Context assembly order: critical information at the extremes

Walkthrough:

1. **System Prompt** is placed first (beginning of context) because it contains the persona, rules, and guardrails that must never be violated. High attention ensures the LLM consistently follows these.
2. **Conversation History** follows, as earlier turns of the current interaction provide continuity.
3. **RAG Retrieved Knowledge** is assembled in the middle with relevant but dynamic content.
4. **Tool Definitions & Outputs** also sit in the middle, containing schemas for available functions and their results.
5. **State & Memory** provides customer data and session state.

6. **User Input** is placed last (end of context) to receive high attention, ensuring the model directly addresses the current query.

> ⚡ **Danger**
>
> If you have five RAG documents placed between the system prompt and user query, document 3 (the middle one) is most likely to be underweighted by the model.

7.4 Mitigation Strategies

1. **Keep the context compact.** Minimize irrelevant tokens so the "middle" is small.
2. **Place critical information at the extremes.** System prompt at the start, user query at the end.
3. **Dynamic content in the middle.** RAG chunks, tool outputs, and other content where minor inaccuracy is tolerable.
4. **Verify tool results independently.** Use separate validation agents to catch any hallucination from middle-positioned content.
5. **Critical guardrails should never be in the middle.** For example, "never write to the database without explicit permission" must be at start or end.

One surprisingly simple technique for boosting attention to key instructions has emerged from recent research, and it requires no architectural changes at all.

8 The Prompt Repetition Effect (Google Paper)

A recent Google paper [4] demonstrated that **repeating the exact same prompt twice** in the context significantly improves LLM output quality, at the cost of doubling token consumption.

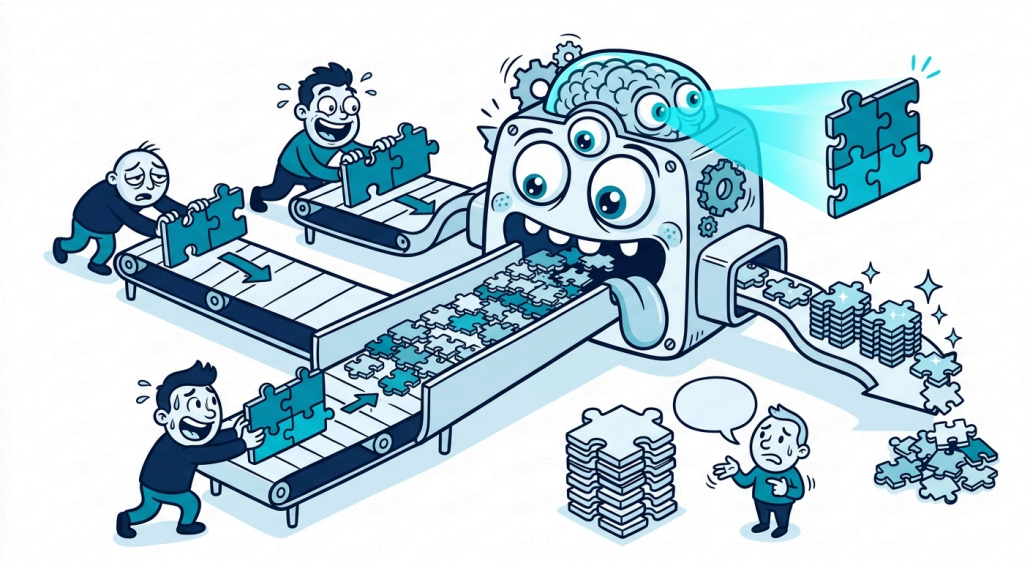

Figure 11: Telling the LLM once: ignored. Telling it twice: "Oh, you were serious?"

Mechanism: The duplicated prompt reinforces the instructions in the attention mechanism, causing the model to attend more strongly to the user's intent. The attention weights are constructed such that when a prompt appears only once, the LLM cannot carry the information as efficiently as when the same tokens appear twice.

Trade-off:

Table 7: Prompt repetition trade-off

Metric	Effect
Token consumption	2x (doubled)
Cost	2x (doubled)
Output quality	Significantly improved

> 🔥 **Practical Tip**
>
> Try repeating your prompt 2x or 3x and observe the quality improvement versus cost increase. This is an easy experiment to run.

With an understanding of context rot, positioning effects, and reinforcement techniques, we are now ready to examine the six distinct sources that populate the context window.

9 The Six Core Elements of the Context Window

The context window is a **finite sequence of tokens** that the LLM processes. These tokens come from six distinct sources, each serving a different purpose. Together, they form the complete input to the model.

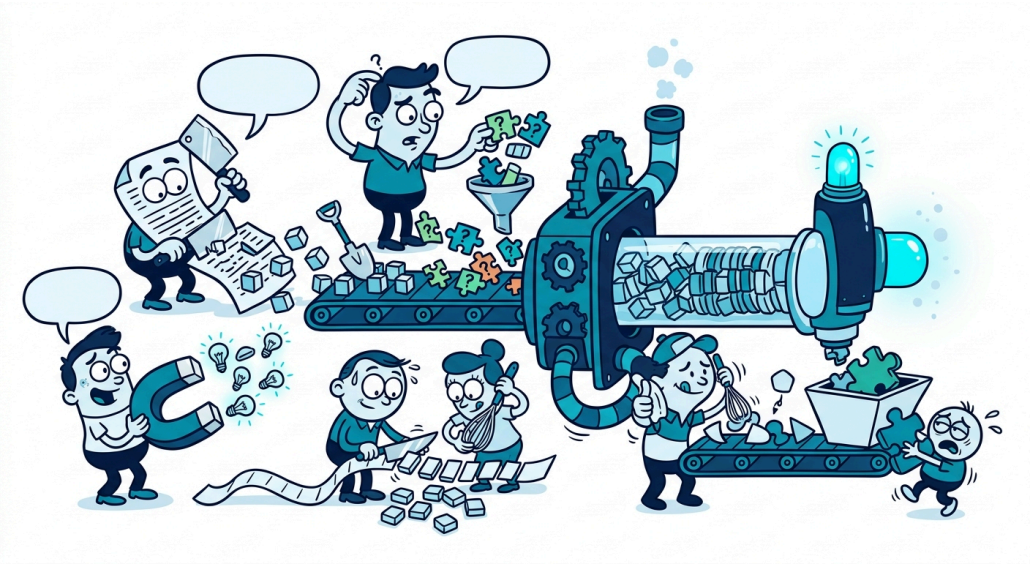

Figure 12: Six ingredients, one context window, zero room for leftovers

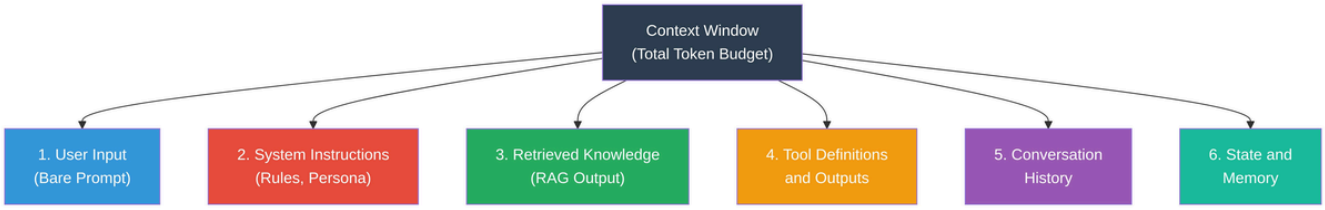

Figure 13: The six core elements of the context window

Walkthrough: The context window has a fixed token budget (e.g., 128K, 200K, 2M tokens). Each of the six elements consumes a portion of that budget. Context engineering is the art of filling this budget with the highest-signal tokens from each element.

9.1 Element 1: User Input (Bare Prompt)

- **What it is:** The current query or instruction from the end user
- **Typical size:** ~100–500 tokens (one paragraph)
- **Position in context:** End (to receive high attention)
- **Key property:** This is the only element the user controls; it cannot be "engineered" by the developer because you cannot ask a customer to learn prompt engineering
- **Example:** "Based on the past conversation, the portal just gave me the same error. I called your support line and sat on hold for 45 minutes before I got disconnected…"

9.2 Element 2: System Instructions

- **What it is:** The developer-crafted instructions that define the LLM's persona, rules, guardrails, and behavioral constraints
- **Typical size:** ~200–2,000 tokens
- **Position in context:** Beginning (to receive high attention)
- **Stored as:** CLAUDE.md files (Claude Code), system prompt field in API calls, markdown files
- **Example:** "You are Alex, a senior resolution specialist at NovaTech Electronics. You have 8 years of experience and authority to approve resolutions up to $2,000 without manager approval. Rules: Always lead with empathy. Use the customer's first name. Never say 'I can't' or 'that's not possible.'"
- System prompt design is covered in depth in the companion guide on system instructions

9.3 Element 3: Retrieved Knowledge (RAG)

- **What it is:** Relevant documents/chunks fetched from a knowledge base via a Retrieval-Augmented Generation pipeline
- **Typical size:** ~600–8,000 tokens (depending on chunk count and size)
- **Position in context:** Middle
- **Pipeline:** Documents → Chunking → Embedding → Vector Store → Query → Top-K Retrieval → Reranking → Context Assembly
- **Example:** Company policy documents, product specifications, return/refund policies
- **Key calculation:** If 8,000 tokens are allocated for RAG and there are 10 chunks, each chunk is ~800 tokens ≈ ~1 page of text

9.4 Element 4: Tool Definitions and Outputs

- **What it is:** JSON schemas describing available tools (functions the LLM can call) and the results returned from those tool calls
- **Typical size:** ~200–500 tokens for definitions; variable for outputs
- **Position in context:** Middle
- **Format:** Structured JSON, not paragraphs
- **Example tools:** `lookup_order(order_id)`, `process_replacement(order_id, reason)`, `issue_refund(order_id, amount)`, `issue_service_credit(customer_id, amount)`
- **Via MCP (Model Context Protocol):** Standardized way for LLMs to interact with tools (e.g., Slack MCP, database MCP, finance MCP)

> ⚠ **Warning**
>
> Having 50+ tool descriptions in the context can bloat it significantly. Tool descriptions alone can overwhelm the rest of the context.

9.5 Element 5: Conversation History

- **What it is:** Previous turns of user–assistant interaction within the current session
- **Typical size:** ~500–10,000 tokens (grows with conversation length)
- **Position in context:** Between system prompt and user input
- **Format:** Alternating `user:` / `assistant:` messages
- **Key issue:** After 15+ turns, the accumulated history can dilute the system prompt, causing the bot to lose its persona and become "blunt and robotic"
- **Solution:** Compaction. Summarize older turns and keep recent turns verbatim.

9.6 Element 6: State and Memory

- **What it is:** Persistent and session-specific information about the user, their history, preferences, and current situation
- **Typical size:** ~200–1,000 tokens
- **Position in context:** Middle (before user input)
- **Example:** "Customer: Marcus Chen. Tier: Platinum (lifetime spend: $12,847). Satisfaction level: at-risk. Current issue: defective headphones, $189.99."
- **Two types of memory:**
  - **Persistent memory:** Long-term preferences, writing style, summarized email history (stored as markdown files)
  - **Short-term memory:** Recent interactions, current session state (stored as JSON or markdown)

Now that we have surveyed all six elements, how do leading practitioners define the discipline that ties them together?

10 Definitions of Context Engineering

Context engineering is still a young discipline, and its boundaries are being shaped in real time by the practitioners building on top of LLMs. Three influential definitions have emerged, each capturing a different facet:

Table 8: Definitions of context engineering

Source	Definition	Emphasis
Andrej Karpathy (former OpenAI)	"The delicate art and science of filling the context window with just the right information for the next step."	Art + science; minimalism; step-by-step
Toby Lutke (CEO, Shopify)	"The art of providing all the context for the task to be possibly solved by the LLM."	Completeness; task-centric
Harrison Chase (CEO, LangChain)	"Building dynamic systems to provide the right info and tools in the right format."	Systems engineering; dynamic assembly; format matters
Anthropic (key principle)	"Find the smallest set of high-signal tokens that maximize the likelihood of your desired outcome."	Minimalism; signal-to-noise; outcome-focused

These definitions converge on one theme: context must be engineered, not improvised. To see what that looks like in practice, we turn to memory, the layer that gives agents long-term continuity.

11 Memory Architecture Patterns

State and memory, the sixth core element, deserves deeper treatment because it is where most production agents differentiate themselves. A well-designed memory layer gives the agent continuity across sessions, awareness of user preferences, and access to historical context without overwhelming the token budget.

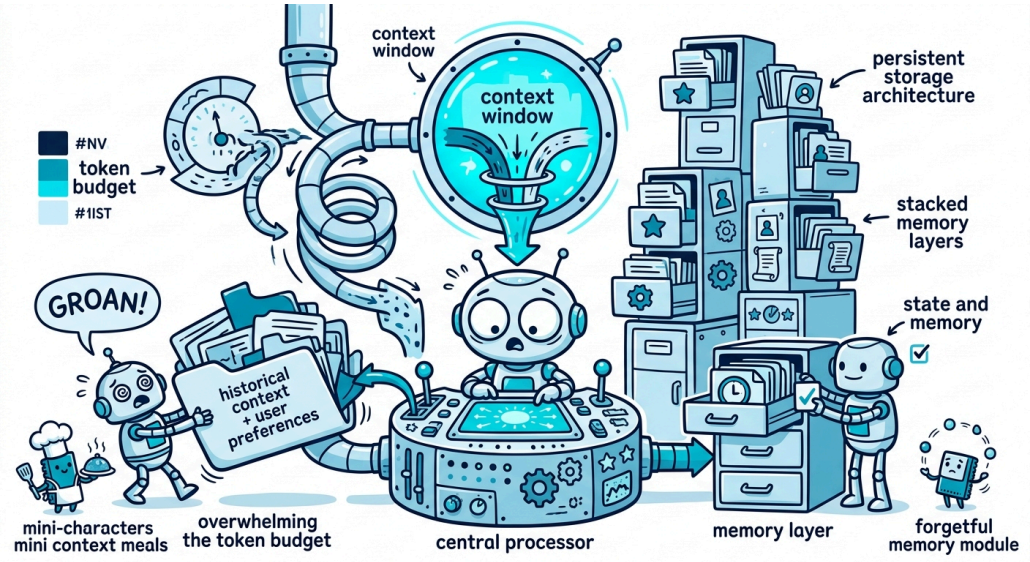

Figure 14: Giving your AI agent a memory so it stops asking "who are you?" every conversation

11.1 Storage Formats

Memory in LLM applications is stored primarily in two formats:

Table 9: Memory storage formats

Format	Use Case	Example
Markdown (.md)	System prompts, rules, preferences, summaries	CLAUDE.md, memory.md
JSON	Structured data (email threads, customer records)	email_history.json
XML	Some system configurations	Older format, less common now

> **Idea**
>
> **Common pattern:** Store structured data as JSON, then create a markdown metadata summary of that JSON for context loading. The full JSON is too large for context, but the markdown summary fits.

11.2 The Email Agent Memory Stack

The instructor's email reply agent uses a layered memory architecture:

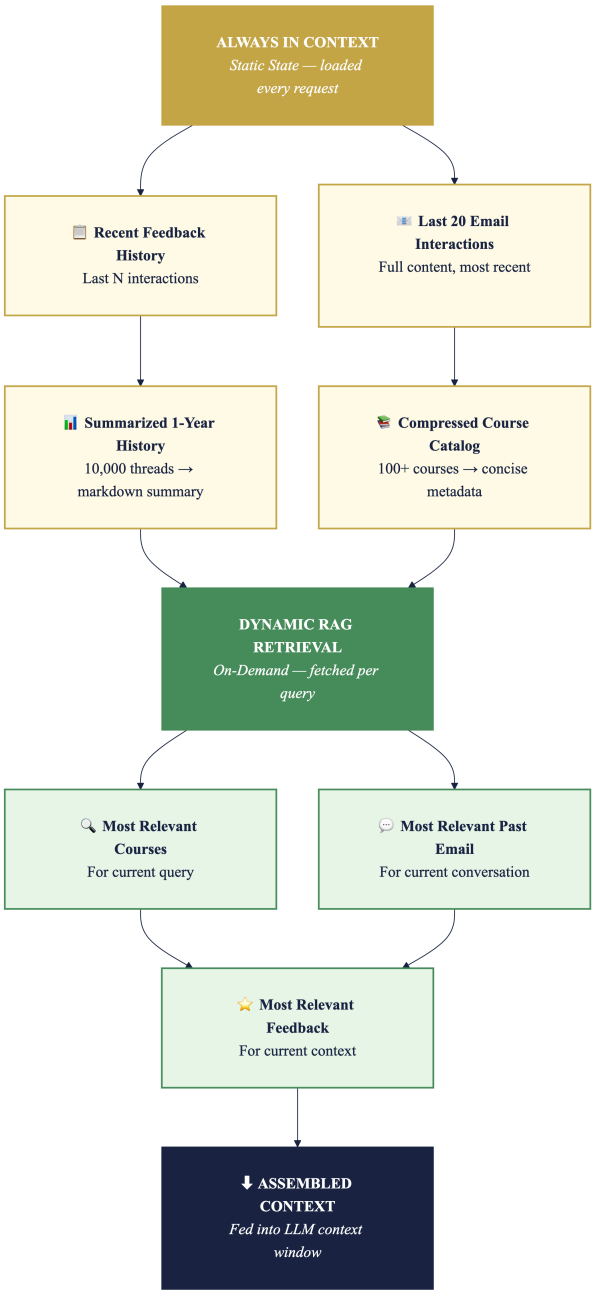

Figure 15: Email agent memory stack: static state plus dynamic RAG retrieval

Walkthrough:

1. **Static state** is always loaded: recent feedback, last 20 emails (high recency value), compressed summary of all historical emails, and compressed course catalog.
2. **Dynamic RAG** fetches additional relevant content on demand: the most relevant courses for the current query, the most relevant past email for conversational context, and the most relevant feedback entries.

3. This two-tier architecture (static state + dynamic retrieval) balances comprehensive context with manageable token budgets.
4. The 250 MB of raw JSON email data is never loaded directly. Only the compressed summary and dynamically retrieved relevant emails enter the context.

With all the concepts, patterns, and principles covered, let us consolidate the most important lessons from this introductory session of Context Engineering.

12 Key Takeaways

- **Context engineering is the successor to prompt engineering.** In 2023, the buzzword was prompt engineering; in 2025–2026, it is context engineering. The user's prompt is just ~6% of the total context — the other 94% is what you engineer.
- **The context window is the LLM's RAM:** volatile, finite, and must be managed carefully regardless of size.
- **There are six core elements of the context:** user input, system instructions, retrieved knowledge (RAG), tool definitions/outputs, conversation history, and state/memory. Each progressively improves output quality.
- **Context rot is the central enemy.** Output quality degrades with excessive tokens, even before the window is full. The Anthropic principle applies: find the *smallest* set of high-signal tokens.
- **The "lost in the middle" effect** means critical information (system prompts, guardrails) must be placed at the beginning or end of the context, never buried in the middle.
- **Repeating the prompt twice** (Google paper) significantly improves output quality at the cost of doubling token consumption, a simple but effective trade-off.
- **The sweet spot for a 200K context window is approximately 50K–100K tokens,** enough for comprehensive context without rot, with room for generation output.
- **Sub-agents get isolated context windows.** When using multi-agent systems, each agent should receive only the context relevant to its specific task.
- **Memory should be layered:** compressed historical summaries always in context + dynamic RAG retrieval for query-specific information.
- **System prompts stored as markdown files (CLAUDE.md) are the starting point** for any context engineering effort.

For quick reference, the glossary and notation table below consolidate all terminology introduced in this guide.

13 Glossary

Table 10: Glossary of key terms

Term	Definition
Context Window	The total input token capacity of an LLM. All text (system prompt, user input, history, etc.) that the model processes at inference time
Context Rot	Degradation of LLM output quality caused by excessive, irrelevant, or contradictory tokens in the context
Context Engineering	The discipline of designing and managing all information that flows into an LLM's context window to maximize output quality
Prompt Engineering	The practice of crafting a single user text input to an LLM for optimal results
Vibe Coding	Iterative, feeling-based interaction with an LLM to produce code, without explicit programming (coined by Karpathy)
Lost in the Middle Effect	The empirical finding that LLMs under-attend to information in the middle of the context window
Needle in a Haystack	A test evaluating whether an LLM can attend to a single critical token at an arbitrary position in a large context
Token	The fundamental unit of LLM input/output; ~0.75 English words using BPE tokenization
RAG	Retrieval-Augmented Generation. A pipeline that retrieves relevant documents from a knowledge base and includes them in the LLM context
MCP	Model Context Protocol. A standardized protocol for LLMs to interact with external tools and services
System Prompt	Developer-crafted instructions defining the LLM's persona, rules, and behavior; stored as CLAUDE.md in Claude Code
LLM-as-Judge	Using a separate LLM API call to evaluate/grade another LLM's output against defined criteria
BPE	Byte Pair Encoding. A tokenization algorithm that splits text into subword units; standard in modern LLMs
Sub-Agent	An agent spawned by another agent with its own isolated context window for parallel task execution
Context Compaction	The process of summarizing/compressing context to free up tokens while preserving information (/compact in Claude Code)

14 Notation Reference

Table 11: Notation reference

Symbol	Meaning
1 token	~0.75 English words (BPE)
1 page	~300 words
K	Thousand (e.g., 128K = 128,000)
M	Million (e.g., 2M = 2,000,000)
Top-K	The K highest-ranked items from retrieval

15 Open Questions / Areas for Further Study

1. **Why does the lost in the middle effect occur mechanistically?** Attention mechanisms are designed for long-range dependencies, yet empirically, middle tokens are underweighted. No satisfying theoretical explanation exists.
2. **How does the quality curve extrapolate beyond 200K tokens?** The sweet spot of 25–50% is empirical for ~200K windows. Does this ratio hold for 2M or 10M token windows? Experiments are needed.
3. **Prompt repetition: why does it work?** The Google paper shows repeating prompts 2x improves quality, but the mechanism (attention reinforcement?) is not fully understood.
4. **How to systematically optimize context assembly order?** Beyond the heuristic of "critical info at the edges," is there an automated way to find the optimal ordering for a given task?
5. **Context rot measurement:** How to quantitatively detect context rot in production systems before output quality degrades? The Chroma article provides some metrics, but standardized benchmarks are lacking.
6. **Read the following recommended papers/articles:**
   - "Lost in the Middle" paper (2023, ~3,500 citations). Search "lost in the middle LLM"
   - Chroma's article on context rot. Search "chroma context rot"
   - Google's prompt repetition paper (2025). Search "repeating the prompt twice Google"
   - Matt Schumer's viral article on AI changes (~85M impressions)
   - Needle in a Haystack blog article

References

[1] A. Karpathy, "The LLM OS."

[2] Chroma, "Context Rot in Large Language Models."

[3] N. F. Liu *et al.*, "Lost in the Middle: How Language Models Use Long Contexts," *Transactions of the Association for Computational Linguistics*, vol. 12, pp. 157–173, 2024.

[4] D. Zhou and others, "Prompt Repetition Improves Language Model Output Quality," *Google Research Technical Report*, 2025.

Follow me for More AI Content

If you found these notes useful, connect with me on LinkedIn for more deep dives into Machine Learning, Artificial Intelligence, and Computer Vision.

Isham Rashik on LinkedIn

Scan the QR code or click the link above