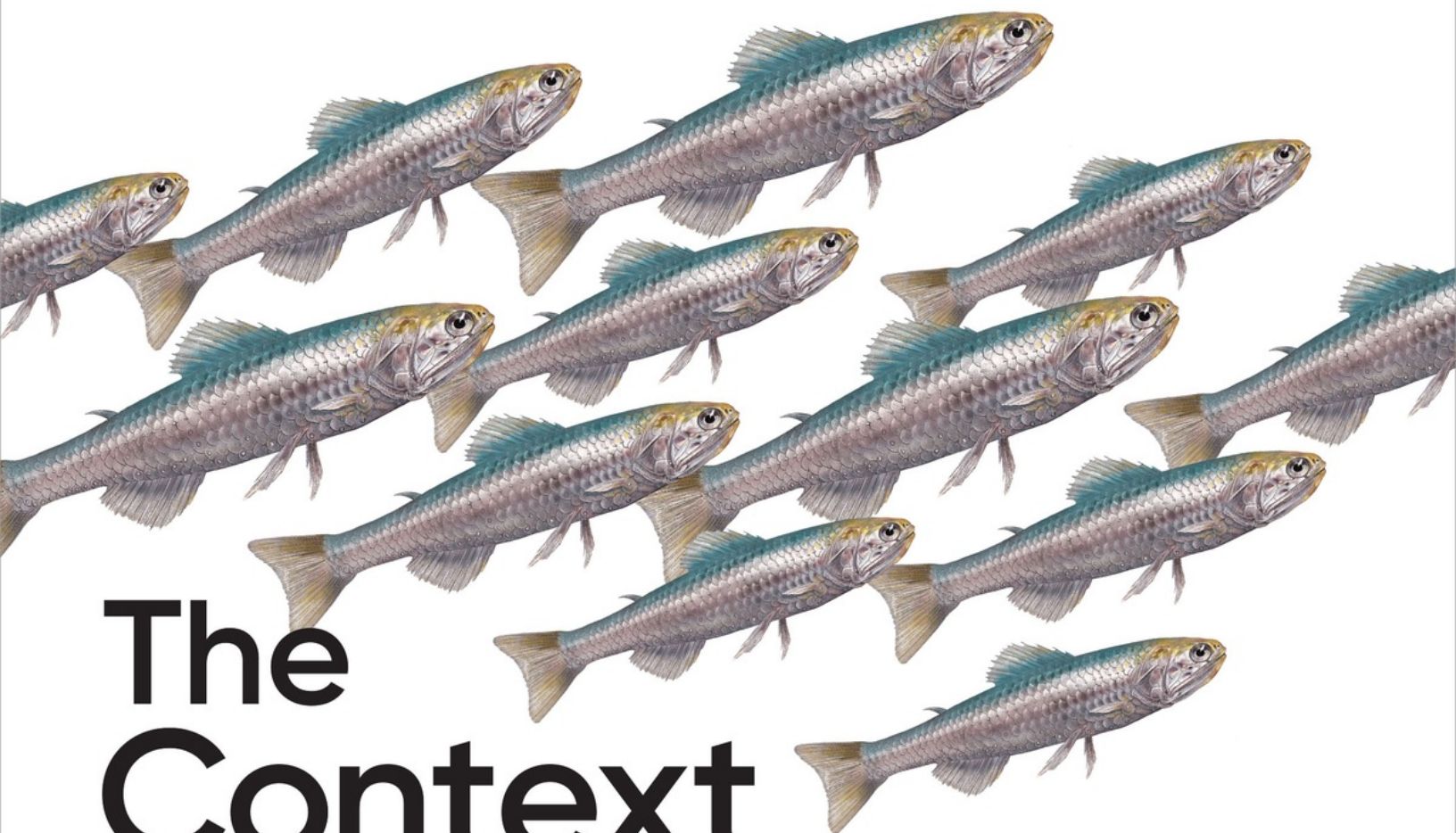


O'REILLY®



The Context Engineering Handbook

The Art & Science of
Structuring Information for AI

**Early
Release**

RAW &
UNEDITED

Edited by Drew Breunig

The Context Engineering Handbook

The Art & Science of Structuring Information for AI

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Edited by Drew Breunig

O'REILLY®

The Context Engineering Handbook

Edited by Drew Breunig

Copyright © 2027 O'Reilly Media, Inc. All rights reserved.

Published by O'Reilly Media, Inc., 141 Stony Circle, Suite 195, Santa Rosa, CA 95401.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<https://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Acquisitions Editor: Andy Kwan

Development Editor: Michele Cronin

Production Editor: Katherine Tozer

Cover Designer: Susan Brown

Cover Illustrator: José Marzan Jr.

Interior Designer: David Futato

Interior Illustrator: Kate Dullea

June 2027: First Edition

Revision History for the Early Release

- 2026-07-30: First Release

See <https://oreilly.com/catalog/errata.csp?isbn=9798341671041> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *The Context Engineering Handbook*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the authors and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

979-8-341-67101-0

Brief Table of Contents (*Not Yet Final*)

Part I: Introduction

Chapter 1: The Road from Prompt Engineering to Context Engineering, by Mike Taylor (available)

Chapter 2: How LLMs Work: A Mental Model for Context Engineers, by Drew Breunig (available)

Part II: How Contexts Fail

Chapter 3: Context Poisoning (unavailable)

Chapter 4: Context Rot, by Kelly Hong (available)

Chapter 5: Context Confusion (unavailable)

Chapter 6: Fighting the Weights (unavailable)

Part III: How to Fix Your Context

Chapter 7: Data Retrieval (unavailable)

Chapter 8: Context Quarantine (unavailable)

Chapter 9: Context Summarization (unavailable)

Chapter 10: Context Pruning (unavailable)

Chapter 11: Context Offloading (unavailable)

Chapter 12: Eval Analysis (unavailable)

Chapter 13: Prompt Optimization (unavailable)

Chapter 14: Adding Efficiency (unavailable)

Chapter 15: Security (unavailable)

Part IV: Case Studies

Chapter 16: Managing Contexts When Coding (unavailable)

Chapter 17: Hex, by Caitlin Colgrove (available)

Chapter 18: How to Write a Great System Prompt, by Srihari Sriraman
(unavailable)

Chapter 19: Compound AI: Offloading Search to Smaller Models
(unavailable)

Chapter 1. The Road from Prompt Engineering to Context Engineering (and Where We Go From Here)

By Mike Taylor

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book. Please note that the GitHub repo will be made active later on.

If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at mcronin@oreilly.com.

This chapter will deliver a brief history of “prompt engineering” as a practice, explaining how it evolved around single-turn LLM applications and chatbot usage, and connect it to the emergence of “context engineering”. The chapter is based on my experience as an AI engineer from 2020 to today, including anecdotes of what problem each new evolution solved. I’ll introduce you to the technologies and patterns that gave rise to “context engineering”, and how it differs from prompt engineering. We’ll explore a few potential futures, including prompt optimization and managing fleets of agents, and why we’ll never escape the need to clearly communicate what we want AI programs to do.

Origins and Early Days (2018–2021)

The transformer model was conceived in 2017 with the famous “**Attention Is All You Need**” paper, but GPT-2 in 2019 was the first language model that I saw smart people in my network experimenting with and talking about. The model was initially deemed “**too dangerous to release**” by OpenAI, which now seems quaint given what we have access to today. AI copywriting was an early use case, which interested me as someone running a 50-person marketing agency, but the results were too unreliable for serious use.

The release of GPT-3 in June 2020 was the birth of prompt engineering. The word “prompt” had been used in computing **since 1977**, but the phrase “prompt engineering” **was coined** by writer Gwern Branwen in June of 2020, a few weeks after GPT-3’s release. The **GPT-3 release paper** put strong emphasis on prompting, showing that larger language models responded significantly better on tasks with instructions and examples in the prompt (**Figure 1-1**).

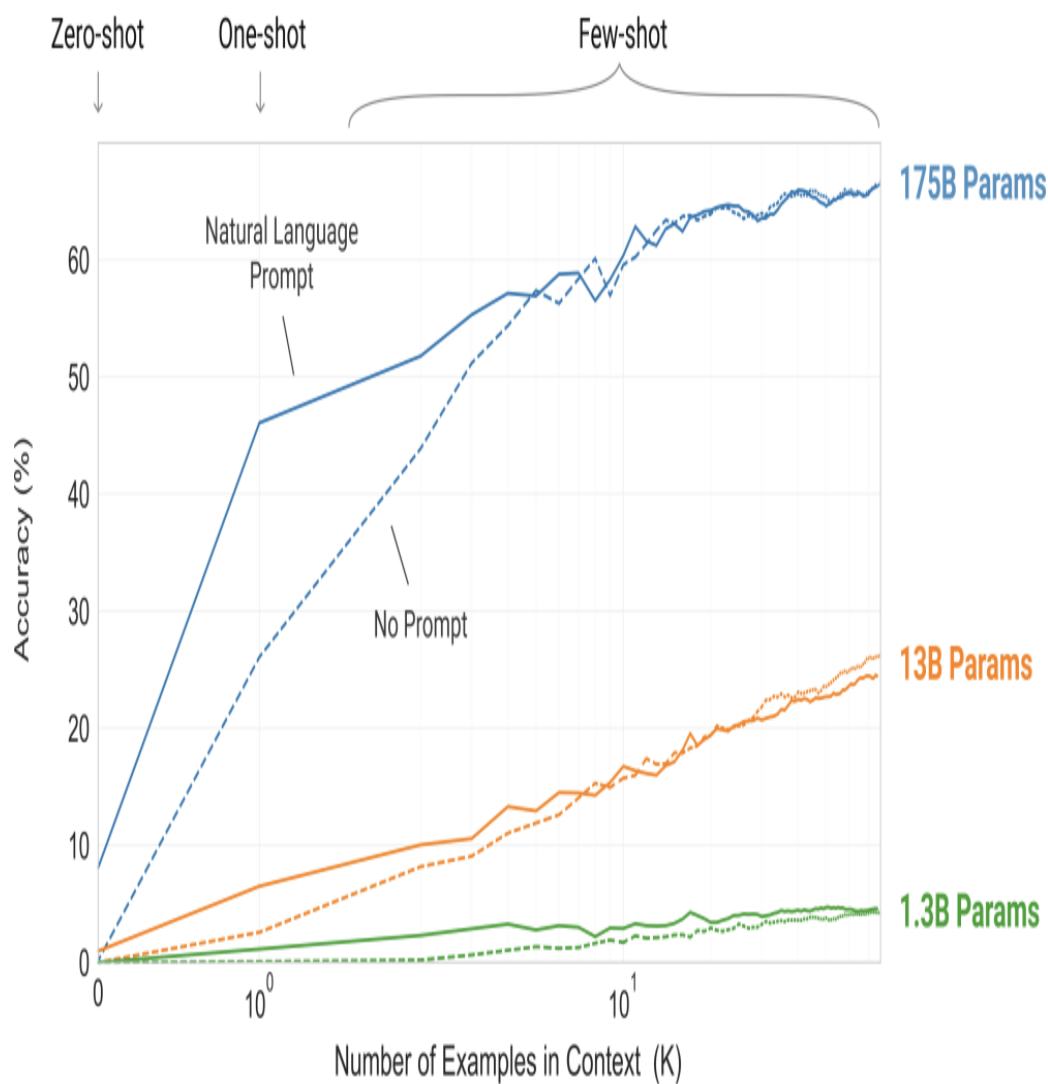


Figure 1-1. Larger models make increasingly efficient use of in-context information. *Source*

Sticking with the “too dangerous to release” story, OpenAI slow-rolled GPT-3 access, making it only available to researchers initially. I first got access through Copy.ai, after the founder Paul Yacoubian tweeted he would give a free account to anyone who was rejected from YCombinator. I was blown away, but had left my agency in March 2020. I didn’t have immediate use cases, but I started experimenting in my spare time.

GPT-3 (davinci) required a lot of experimentation because it could only handle 2,049 tokens, and was still extremely unreliable. Great efforts were made to find exactly the right combination of words that gave better results, “figuring out how to hack the prompt by adding one magic word to the end that changes everything else” as **Sam Altman described it**. Few-shot learning, or including examples in the prompt, was the dominant technique, but early **RAG (Retrieval Augmented Generation)** methods were developed to bring relevant context using vector search into the prompt at the right time.

It was clear at this point AI would be a big thing, with Andrej Karpathy calling prompt engineering “**Software 3.0**”—the next big evolution in computing since machine learning. Over the next two years I used GPT-3 to automate a lot of the work I was doing, in order to free up more time to play around with AI. I made a conscious effort to try every single task with GPT-3 first, and spend some time prompting to see if I could get it right, before giving up and doing it myself. It was during this time period that I developed the prompt engineering principles my **first O’Reilly book** was based on:

Give Direction

Describe the desired style in detail, or reference a relevant persona.

Specify Format

Define what rules to follow, and the required structure of the response.

Provide Examples

Insert a diverse set of test cases where the task was done correctly.

Evaluate Quality

Identify errors and rate responses, testing what drives performance.

Divide Labor

Split tasks into multiple steps, chained together for complex goals.

This was at a time when most prompts were call and response—most AI users were not doing anything “agentic” and had to be coached to try and break prompts up into multiple steps for better reliability. The human was manually curating the examples, writing out descriptions, and formalizing the evaluation methods, in a copy-pasted prompt template. In testing to improve these prompt templates, I noticed that the same things that worked with prompting GPT-3 mapped to management principles I used when running my marketing agency. Neural networks are based on the function of biological neurons, so it made sense to me that some of the same techniques you use for managing biological intelligence would also help when managing AI.

Image Generation Era (2022)

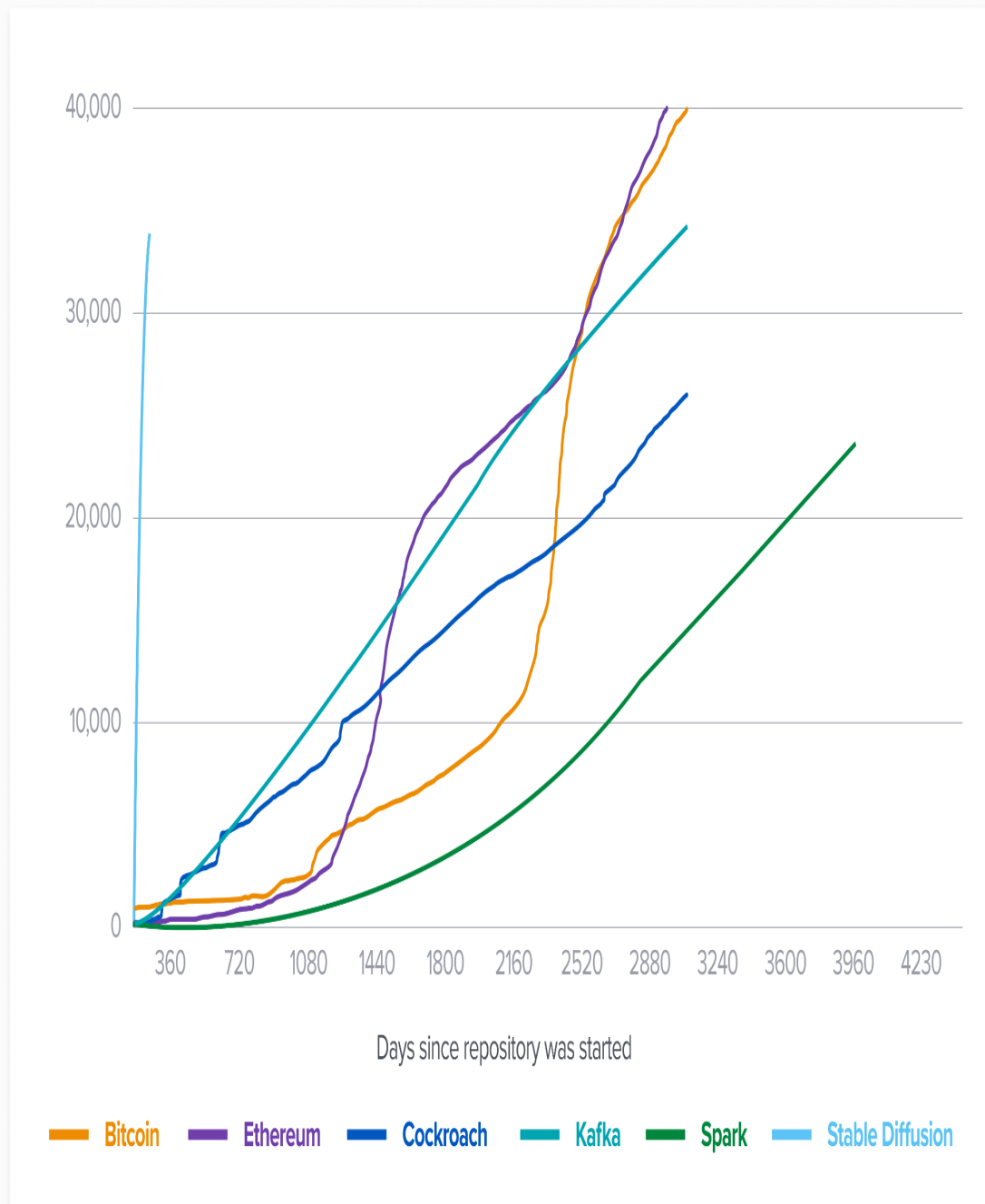
Prompt engineering was still a niche activity for the few people who were experimenting with GPT-3, which was still primarily accessed via an API. It was actually image generation with the launch of DALL-E 2 in April 2022 that brought prompt engineering into the mainstream.

OpenAI had revealed DALL-E in January 2021, but DALL-E 2 was the first text-to-image model to demonstrate more realistic images that combined concepts, attributes, and styles. The famous “**horse-riding astronaut**” image and those like it shared on social media by those with access, drummed up demand. Again, OpenAI slow-rolled access which was only granted to a handful of researchers and industry insiders.

Like millions of others, I got access to Midjourney months before they dropped the waitlist for DALL-E 2. This allowed the small startup to build

a vibrant community centered on Discord (the only way you could use the model at the time). Stable Diffusion was released in August 2022 and became one of the **fastest growing open-source projects** in history (**Figure 1-2**). People were astounded that you could get comparable results to DALL-E 2, running for free on your local computer (if you had a GPU).

Stable Diffusion Developer Adoption



Source: GitHub

Figure 1-2. Stable Diffusion developer adoption, a16z (source)

This was when prompt engineering went mainstream, with millions of people in online communities sharing tips on how to get reliable results out of these image models. In image generation reliability meant two things: 1) getting the subjective style that you wanted, and 2) keeping character and scene consistency between images so they could start to tell stories. This required precision that was hard to capture in plain English for those of us who had no art history background. I remember learning the term “Trompe-l'œil” (a Renaissance painting technique for showing 3D objects on a two-dimensional surface) for the first time, after seeing other people use that effect in the Midjourney Discord (which had millions of people sharing their work by default). Once you know a specific magic word you can use it as shorthand to get that precise style. The key to success was knowing the right sequence of magic words to get what you wanted.

Being able to run the models on their own computer (without paying per token) made a big difference to adoption at a time when you would have to experiment with 10-20 prompts to get a usable image. People started training their own fine-tuned versions of the models to cater to specific tastes, and to avoid **overly strict content moderation from OpenAI**. Of course, like most new technology, people immediately used it to create explicit images, and ironically many of these NSFW models became very good at human shape and form, and were repurposed as the first “photorealistic” models (**requiring negative keywords to avoid mishaps!**). Many innovative techniques like inpainting (replacing part of an existing image), control nets (generating a new image over the same lines) and dreambooth (teaching a new character or style to a model) became widespread in the open-source community long before becoming available in frontier models. This was for many people an important lesson in the value of open-source models, which we took with us when the market refocused on text generation again at the end of the year.

GitHub Copilot became generally available in June 2022, and defined the paradigm for all enterprise adoption of AI tools going forward. It popularized the copilot model, where you collaborate with an AI to get

work done, in this case autocompleting your code as you type. I distinctly remember at the time that I started doing test-driven development, writing more comments in my code and using types religiously, because it would help Copilot do a better job at autocompleting—another sign that the practices we developed for people to collaborate better, would transfer to working with AI.

One hangover from this period is that the term “prompt engineer” started to get polluted by Twitter gurus cashing in on the hype. Early prompt engineers were actual engineers—GPT-3 was only available via API. This latest wave of non-technical prompt engineers were the equivalent of witch doctors or **medieval alchemists**, writing prompts like **spells or incantations**, rather than formally a/b testing to prove what works. This started to give prompt engineering a bad name, with **Sam Altman declaring** *“I don’t think we’ll still be doing prompt engineering in five years”*, in September 2022, continuing to say *“what will matter is the quality of ideas and the understanding of what you want.”*

ChatGPT Revolution (Late 2022–2023)

In November 2023, OpenAI released ChatGPT as a free research preview, which interacted in a conversational way. Up until then, LLMs had been completion models, meaning you would put in your prompt for the model to complete the next series of tokens. ChatGPT was based on OpenAI’s research with **InstructGPT**, where they found ways to fine-tune GPT-3 to better follow instructions and perform natural language tasks. Within 5 days ChatGPT hit 1 million users, and 100 million within 2 months, making it one of the fastest growing consumer products in history (**Figure 1-3**). AI had crossed the chasm.

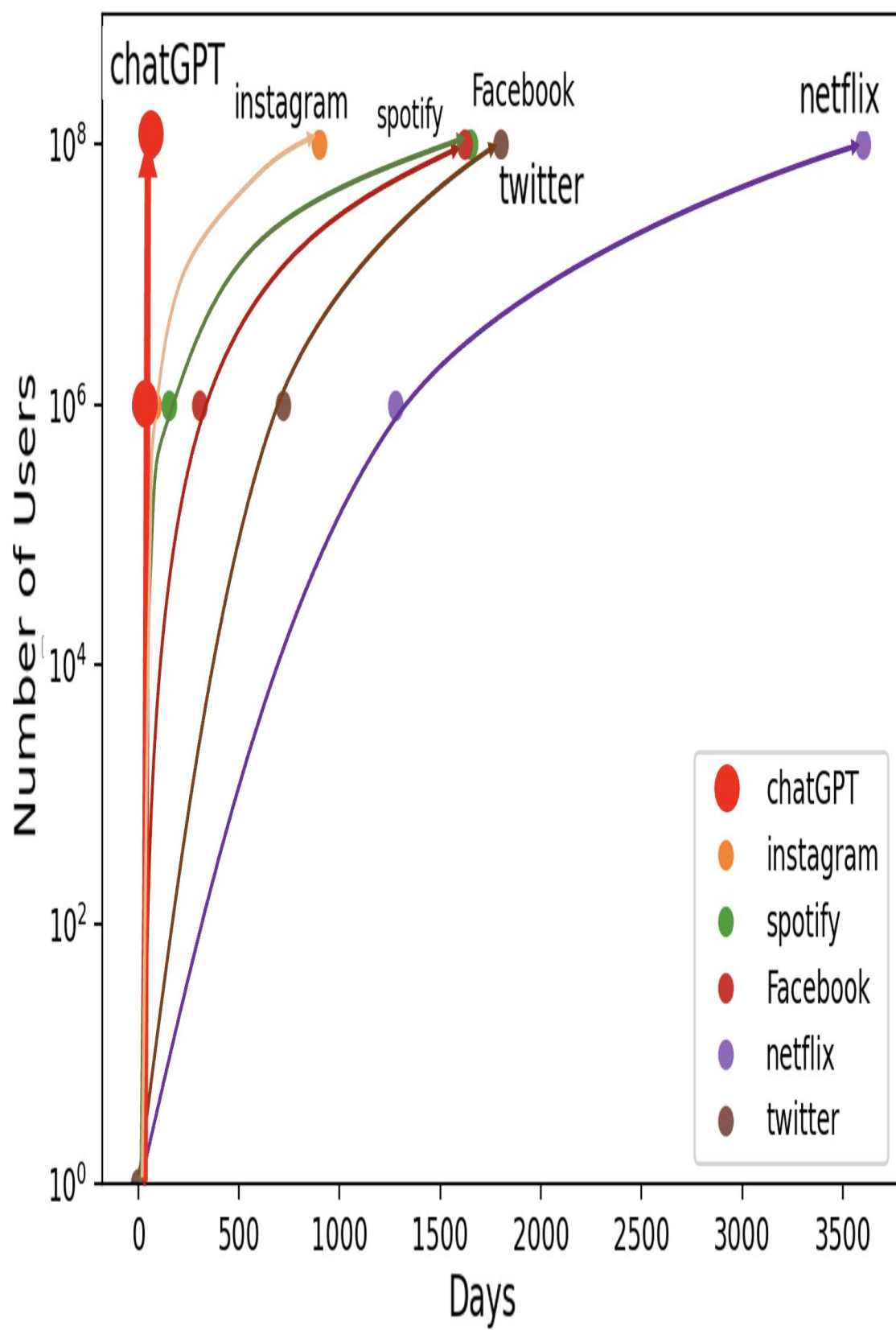


Figure 1-3. ChatGPT: Number of days to 100m user (source)

Up until this point I had been only doing AI for myself—after ChatGPT launched everybody suddenly wanted AI training for their team. Enterprise adoption happened surprisingly quickly: I spoke at an AI conference just a few months after the launch and many large enterprises I spoke to were already rolling out pilot projects. I saw that AI papers published to ArXiv were growing exponentially, **doubling every two years**, and realized that businesses would need help keeping up.

ChatGPT worked much better than GPT-3, particularly when they upgraded the model from GPT-3.5 to GPT-4. However, that just meant we used it for more ambitious tasks, and so reliability remained a problem. Suddenly people were willing to pay me to teach them prompt engineering, as a potential solution to the same issues we saw with image generation around reliability and consistency. It's hard to get good results if you can't describe what you want in the prompt using plain English.

Intense competition in the AI space that continues to this day ignited in 2022-23. Google had invented the transformer, but had been too slow to market, allegedly due to concerns over safety, negative press, and potential disruption of their comfortable market position in search. Management issued a **"code red"** after seeing the success of ChatGPT, bringing co-founders Larry Page and Sergey Brin out of quasi-retirement to release Bard in March 2023, later rebranded to Gemini.

The Anthropic founding team that was formed after splitting from OpenAI in 2021 over safety concerns, were also cajoled into launching their own ChatGPT competitor called Claude in March 2023. At the same time, OpenAI launched GPT-4, which was a significant leap over GPT-3 and GPT-3.5 (the fine-tuned ChatGPT model), and had a 8,192 context window. Anthropic responded with a 100k context window in May 2023, nicknamed **"clong"** or "claude long" by fans of the model. This was the first time I heard **"RAG is dead"**: why would you need to retrieve context from a database when you could fit 100k tokens or 75k words in the prompt?

Despite huge adoption, GitHub Copilot failed to develop noticeable new features, leaving the door open to Cursor, which launched in March 2023 and iterated quickly to several innovations. My personal favorite was “yolo” mode, where the agent could edit multiple files at once, which was terrifying at the time for those of us used to tab autocomplete.

Early Agent Era (2023–2024)

Although published in October 2022, the **ReAct (Reason and Act) pattern** started to pick up steam in 2023, as function calling or tool calling began to be supported in the summer by OpenAI. Most AI agents today are an evolution on this foundation, where you prompt an LLM to think, then take an action, before observing the outcome and repeating the loop. Calling external tools with pre-determined code helped AI give more reliable results, for example being able to search first before answering a question. LangChain, an LLM orchestration framework, began to be widely popular as people looked to write more transferable code across models and workflow patterns.

Despite predictions that prompt engineering was going away as models got smarter and context windows got longer, it became more important than ever. Greg Kamradt’s famous **“needle-in-a-haystack”** test found that AI models didn’t uniformly pay attention to all the information in long prompts, dispelling the myth that “RAG is dead” because you could just put everything in the context window. Additionally, GPT-4 was **15-30x more expensive than GPT-3.5**, and so if you could get your workflow running on a smaller, cheaper model with better prompting, it was worth the effort. With Meta’s release of the Llama model series, it was suddenly possible to get GPT-3 level models running for free locally on your computer—the **“stable diffusion moment”** for language models, as British programmer Simon Willison put it.

The hype around prompt engineering exploded around this time, with Anthropic advertising a new role for “Prompt Engineer and Librarian” with an attention-grabbing salary of \$250k-\$335k. People start asking if prompt

engineer will be a **job of the future**, which is a bit like asking if you should start calling yourself an Excel engineer. I actually was calling myself a prompt engineer at the time, and was working full time freelancing as well as growing my **Udemy course** which launched in April 2023.

There are two paths people took to solving the reliability problem with LLMs:

- Carefully define the structure to your prompt pipeline and optimize it until it works reliably against a formal evaluation metric, even with smaller less intelligent (and less expensive) models.
- Throw more compute at the problem, with smarter (and more expensive) models, using agentic workflows and tool harnesses to aid the model in staying on the right trajectory to reach a solution.

Both approaches apply Richard Sutton's **Bitter Lesson**—that more compute eventually beats human expertise—in different ways. Carefully defining inputs and outputs allows you to treat the prompt as a 'black box', and benefit from gains in model quality, optimization techniques, or inference strategies. Throwing increasingly powerful models at the problem inflates costs in the short-term, but as LLM inference costs drop **9x-900x annually** for the same level of intelligence, frontier capability quickly becomes too cheap to meter. Most prompt engineers started calling themselves **AI engineers** around this time, with those from startup backgrounds typically opting for agentic workflows, and those from large company data science backgrounds being naturally attracted

For those in the first camp, in October 2023 Omar Khattab launched **DSPy**, a framework that promised “programming—not prompting LMs”, in a move away from natural language and towards structured precision. The library abstracted away brittle prompt strings instead using “signatures”, defined inputs and outputs with types, leaving the prompt instructions to be learned by an optimizer based on performance against your dataset and evaluation metric. I found out about the library when someone saw my badge that read “prompt engineer” at an AI conference and gleefully reported “don't you know prompt engineering is dead? Check out DSPy”,

sharing **this article** with me. It turns out that DSPy still required collecting an example dataset, defining a formal evaluation metric, and deciding what inputs should go in context. Writing the prompt was never the hard part of prompt engineering, in the same way that building a bridge is less complicated than designing it.

Those in the second had their time to shine in early 2024, with the rise of the vibe coding (generating code without looking at it) era, as GPT-Engineer rebranded to Lovable, Replit trained their first agent, and Codeium rebranded to Windsurf. There were also early experiments with longrunning agents like AgentGPT and BabyAGI, as well as multi-agent systems like Microsoft Autogen and Camel. Vibe coding seemed to suddenly work all of a sudden once Claude Sonnet 3.5 came out in June 2024, even if the term wasn't **coined by Andrej Karpathy**, former AI lead at Tesla, until early 2025. One memorable glimpse into the future was visiting San Francisco and watching an engineer use Devin by Cognition, the \$500 a month autonomous employee (ChatGPT cost \$20/m at the time!), with its own coding environment and Slack access like a regular employee. These early experiments hinted at a future where AI would run for longer on more ambitious tasks in the background without requiring input at every turn.

Context Engineering Paradigm (2025–2026)

As models got more powerful, they were completing longer running tasks, but would eventually go off the rails if their context window filled up too much. The solution wasn't just more prompt engineering, because command-line coding agents like Claude Code and Codex were completing tasks without waiting for you to prompt them. AI engineers had to zoom out and look at the overall system, to curate what context went to an AI agent at any given time. Too much context and they get confused or distracted. Too little context and they can't do a good job on the task. Ankur Goyal of Braintrust **coined the term** "context engineering" in March 2025 to give a name to this new focus area.

Later in June 2025, **Toby Lütke (Shopify)** and **Andrej Karpathy (Tesla, OpenAI)** publicly endorse context engineering as the dominant paradigm. Around the same time the term “**context rot**” arose to describe how LLM performance degrades as the context window fills up, and Drew Breunig listed the different types of **long context failure** in more detail. The **Gemini 2.5 paper admitted** that performance degrades after 200K tokens despite having a nominal context window of over a million tokens. RAG vendors were delighted with this change of fortunes, and Chroma quickly dropped a **technical report** highlighting the issue further. It turns out RAG wasn’t dead after all.

Anthropic calls context engineering “**a natural progression of prompt engineering**” that became important once we had long-running non-deterministic agents that required better context management than was necessary in defined workflows. That said, the word “prompt” is somewhat synonymous with “context” in AI—the prompt is the context window. Even with GPT-3 good prompt engineers were dealing with context rot (it just happened at 2K tokens not 200K), retrieving relevant context to put in the prompt with RAG and other methods, and making tool calls with the ReAct pattern.

My cynical take is that AI engineers were desperate for a less polluted term to differentiate themselves from non-technical prompt engineer twitter gurus. With the backing of Karpathy and Lutke, context engineering filled that role. Anyone who speaks English can prompt, but you need to code to manage context dynamically in AI agents. This semantic shift is similar to the rise of Data Science vs Business Analysts (to distinguish Python coders from Excel users), and Growth vs Marketing (a focus on growing through changes to the product, rather than spending money on advertising). Whatever we call it (and it may well change again!), the skill of testing what works to get more reliable AI results continues to be of paramount importance.

Parallel Coding Agents (2026)

What's better than one long-running agent? Multiple long-running agents in parallel. Naturally, once people realized Claude Code was good enough to complete most tasks with minimal prompting, they wanted to see many tasks they could handle at once. This experimental behavior forced new productivity tools to be needed, such as AI-native task trackers such as **Beads** by Steve Yegge, and cross-agent communication tools like **Agent Mail** by Jeff Emanuel, who was profiled as **running twenty-two Claude Max plans at a time**, costing over \$4,600 a month. Projects like **Gas Town**, where AI managers coordinate AI workers, continue to proliferate as people push the limit of what's possible. Eventually we'll see entire organizations of AI agents, simulating the work of an entire company, as is the vision of Elon Musk's **MacroHard** project (an attempt to replicate Microsoft with AI agents). In the space of a year we'll go from you prompting AI, to AI prompting you.

Being less directly hands-on with AI agents means you have to retreat to surfaces of control that managers have traditionally used to get the most out of their employees. Activities like writing standard operating procedures (SOPs) and documentation as well as defining company culture make a big difference in human-led organizations, and there are AI equivalents. Your `agents.md` file sets the culture and coding style for your repository, MCPs are how agents access tools, and skills are the SOPs of the agentic world, which detail how to do specific recurring tasks. Projects such as OpenClaw (formerly Moltbot, and Clawdbot) add to this a `Soul.md` document serving as the top-level identity layer, as well as various memory files and a `heartbeat.md` for tracking recurring periodic tasks.

Skills are where the traditional AI workflows of the type we built in LangChain, DSPy, or N8N can now live and be used by true AI agents, marrying the two domains of optimized workflows versus vibe coded agents. If your task is only going to be done once you don't need a skill, but if it's going to be done often it makes sense to optimize that workflow and make it more deterministic. If it's going to be done many more times than that, formalize it further into code and make it into an MCP that can run cheaper and faster than waiting for an agent. Expect more focus on prompt

optimization tools like DSPy in training AI skills, as well as workflows that actively learn how to do tasks better each time like Kieran Klaassen's **Compound Engineering** plugin.

If we play the bitter lesson forwards, we should expect that eventually these harnesses fall away and the model no longer needs them to do a reliable job. The popularity of the “Ralph Wiggum” technique — named for the Simpsons character’s “**naive persistence**” — highlights that with smarter models simple approaches still work. The technique consists of a small bash script that pipes a prompt.md file into your coding CLI agent of choice (typically Claude Code), and instructions to take one task at a time, commit with git, and manage a todo list and a progress file locally. This causes a “self-referential feedback loop” where Claude can see its previous work, and then move the project along one cycle at a time until one of the Claudes confirms the task is done. This simple but effective loop started working well once Claude Opus 4.5 came out, allowing 24/7 autonomous agents running as cron jobs in the background AFK (away from keyboard), without a human in the loop, or even designing the harness or environment the agent operates in.

Assuming code generation is (or will soon be) a solved problem, people have started experimenting with spec-driven development, such as **Drew Breunig's whenwords library**, which only implements the specs and tests and lets the user compile the program with their coding agent of choice. Now imagine we snap our fingers and given a spec, the coding agent can build anything we desire... what do you want to build? The specificity problem remains, and we still have to get good at prompting or defining inputs and outputs formally. When you can build anything, the new bottleneck becomes knowing what to build.

AGI Approaches (2027–2029)

Since 1999, Ray Kurzweil has **maintained that 2029** is the year computers will achieve AGI or artificial general intelligence. This is a fuzzy goal that refers to a type of artificial intelligence that possesses the ability to

understand, learn, and apply knowledge across a wide range of tasks at a level comparable to that of a human being. I like to think of it as a coding agent that writes its own specs—understanding enough about the world to make a tasteful contribution to it. Dario Amodei from Anthropic **expects AGI to come in the next 1-2 years**, and Sam Altman **also expects AGI before the end of the decade**. While nobody knows if AI development will continue at the same pace, or if we get to AGI what will happen, we can hazard a guess based on current experience with AI.

The looming potential of AGI on the horizon has led to murmurs of the “permanent underclass” theory circulating in Silicon Valley—the idea that if you’re not aggressively adopting AI right now, you’ll be permanently priced out of the future economy and will have to hope for UBI. The argument has a seductive internal logic: someone on the free tier of ChatGPT is falling behind the person paying \$20 a month, who’s falling behind people like me spending \$200 a month for 20x the usage and access to the most capable models, who in turn are falling behind people spending thousands running multiple coding agents. **Sam Altman plans** to release PHD level models at \$20,000 a month—and if you haven’t learned to be productive at the \$20 tier, you’ll never justify the \$200 tier, let alone the \$20,000 one and beyond. The theory says this ladder pulls up behind you, and the gap compounds until it’s permanent.

That isn’t what we’ve seen with previous technology shifts, however. As the price of intelligence comes down, it’s likely we’ll demand much more of it, a phenomenon known as Jevon’s paradox. In addition, we’ve seen as we experience growth in one fast moving industry, other slower moving more regulated or physically constrained industries trend to increase in price, wiping out some or all of the productivity gains, called Baumol’s cost disease. **Alex Danco of a16z** detailed how the combination of these two phenomena could sustain wage growth even as AI automates the majority of what we currently do for work. If AI eliminates 95% of your job, there’s a chance you’ll just be expected to do 20x more work rather than being eliminated. We haven’t seen a mass extinction event in Radiology, a field

once deemed **highly susceptible to AI automation** that now employs more Radiologists than ever.

Every technology wave from the printing press to the internet has triggered the same panic, and every time Baumol's cost disease spreads the gains around: when one sector booms, wages rise everywhere because everyone competes in the same labor market. Electricians are already commanding half-million-dollar salaries to wire data centers. If AI makes everything it touches a thousand times more efficient, the relative value of human-bottlenecked work goes up, not down. Hairdressers, therapists, tradespeople, and yes, the engineers navigating that messy middle where you have to extrapolate from incomplete data—these roles become the premium goods in an AI-abundant economy.

In my personal experience with AI, I've found that while they can quickly achieve impressive benchmarks on known tasks with optimization, they fail miserably when generalizing to unknown tasks. Ask a coding agent to create a crud app, the likes of which have been seen a trillion times in the training data, and it does a great job. Ask it to do something it hasn't seen before, and it often performs less impressively. This is because AI models work best when there is a lot of data to synthesize, or a lot of examples to follow, but not when you have to make decisions under uncertainty with limited information. AI might be the General, monitoring more data in realtime than any human could, and it might be the soldier, executing common tasks flawlessly every time, but maybe there's room for humans in the **NCO role**: translating the commander's intent to the conditions on the ground, under conditions of uncertainty.

The people who learn to work effectively with AI—who develop taste for what to build, who understand how to engineer context, make the right bets and leaps of faith, and manage fleets of agents—will have a meaningful advantage over those who don't. But that's always been true of new tools. The gap between someone who can use a spreadsheet and someone who can't was once a career-defining edge. The truth is we'll always need prompting skills, whether we call it that or not. Think about the smartest person you've ever worked with, and ask yourself... did they not need HR,

Legal, and Management support because they were so smart? The chances are they needed it more, and these things are all just prompts for humans to align us to the goals of the organization. Because intelligence is becoming an inexpensive commodity, the premium human skill is no longer writing code or generating text, but rather managing the architecture of context that orchestrates these systems.

The skill of clearly communicating what you want, whether to a human or AI agent, isn't going away anytime soon. It's becoming the whole game.

Chapter 2. How Training Shapes LLMs and Explains Their Quirky Behaviors

By Drew Breunig

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at mcronin@oreilly.com.

In late 2025, I found myself struggling to convince GPT-5.1 to properly call a popular open source library. With frustrating consistency, the model would either omit a required attribute or append options that didn’t exist. After the 5th correction, I pulled up an IDE and started hand coding the calls. Perhaps GPT-5.1 simply wasn’t *that* good, I thought. Perhaps I’d be returning to Claude sooner than expected.

But while it appeared GPT-5.1 was hallucinating, the cause behind these hiccups was much more mundane: the model was never trained on the correct API calls. GPT-5.1’s cutoff date was September 2024. Articles, blog posts, code libraries, and other artifacts created after this date weren’t in GPT-5.1’s training data. And the library I wanted to call had significantly

updated its APIs in 2025. GPT-5.1 was writing the correct calls, given its training data.

This problem is mundane and obvious, but its explanation is a general rule that works on almost *every* large language model quirk: *stubborn and unexpected behavior can be explained by understanding how the model was trained.*

A model's inability to count the letters in a word? Its continual flattering of users? Its unrelenting preference for em dashes? That's how it was trained. From mundane issues like diminishing performance as contexts grow to spooky statements that hint at sentience: it all comes back to how the model was trained.

In 2025, **researchers at OpenAI argued** that models output hallucinations, “producing plausible yet incorrect statements instead of admitting uncertainty,” because of the way we train them. “Language models are optimized to be good test-takers, and guessing when uncertain improves test performance.” Only by adjusting how training occurs and how benchmarks are designed can this behavior be fixed.

For those of us not training models, the move is to build a working knowledge of how models work and how they are trained in order to diagnose, mitigate, and/or avoid undesired behavior.

How an LLM Processes Your Prompt

Let's try to build the smallest possible model of a model. No math, no diagrams, and as little jargon as possible.

At the highest level, when you hit enter in ChatGPT, Gemini, or Claude, the following happens:

1. The words you typed are chopped into **tokens**, chunks of characters drawn from a fixed set of a couple hundred thousand strings.

2. Each token maps to an **embedding vector**, a long list of numbers encoding each chunk's semantic meaning.
3. Each embedding is adjusted based on the other chunks around it, modifying the meaning based on its local context.
4. The final sequence of adjusted embeddings is passed through the LLM's **weights**, billions of numbers fixed during training, to score every possible token with a probability that it comes next.
5. One token is selected from these scores, added to the sequence, and the whole process repeats.

On the whole, this process is pretty simple. Confusion comes from the jargon: tokens, embeddings, and weights. Let's tackle each of these in order.

Tokens

The paragraph you entered in the chatbot text field is chopped into chunks of characters called **tokens**. Each model has a fixed vocabulary of tokens, each representing a unique character or characters. GPT-5-class models have a token vocabulary of roughly 200,000 tokens.

A frequent question is, "Why don't we just use one token for each character or each word?" The number 200,000 feels somewhat arbitrary, as do some of the chunks each token represents. For example, here are some bits of text that each map to one token in GPT-5:

- A
- B
- b
- !
- 你
- 4

Makes sense. We're mapping uppercase and lowercase characters, punctuation, characters in multiple languages, and digits.

But here are some other bits of text that are each precisely one token:

- soda
- Yankee
- 965
- INFRINGEMENT
- scipy
- —>

At first, it's hard to find rhyme or reason for *why* some strings get their own token and others don't. But it's a popularity contest: when building a token vocabulary, teams use algorithms to select the most common chunks of characters in a given dataset. They do this to make everything *cheaper*.

All of the costs of LLM are denominated in tokens. If we can feed fewer of them into models during training and inference, models cost less to build and run faster and cheaper. By smartly selecting *popular* tokens, we can represent most inputs with the fewest tokens possible. We say, "Hello", more often than, "Hey there," so the former gets one token while the latter phrase requires three. The common phrase, "Once upon a time," uses the same number of tokens as the nonsense string, "w`i<0!" We strategically select our chunk vocabularies to minimize the amount of tokens we need to process.

But the efficiency this chunking delivers isn't a free lunch; converting multiple characters into a single placeholder that the model sees yields some weird artifacts. Andrej Karpathy, cofounder of OpenAI, argues that much of, "**oddness with large language models can be traced back to tokenization.**"

For example, because the model only sees one token when presented with the word, "Yankee", how could it count the number of vowels in the word?

Tokens and chunking are the culprits behind the notorious, “How many r’s are in the word ‘Strawberry’?” question that plagued model providers for years.

Tokens are the reason LLMs struggle with math. The number 38425 maps to tokens corresponding to “384” and “25”. If I want to add that to 4239 (two tokens), models resort to rote memorization and, once certain model sizes are reached, learn statistical shortcuts.

NOTE

Dive Deeper: Huggingface has an excellent [deep dive into the history of number tokenization](#) that I recommend if you’re curious.

If 200,000 tokens make LLMs more efficient, why don’t we use a vocabulary of 2 million tokens? Or 200 million?

If we use too many tokens, model makers run the risk that some of the tokens are underutilized during training. They accrue little information during the training run, which can result in very weird behavior. For example, [asking GPT-3 to explain the word “SolidGoldMagikarp” causes it to explain what “distribute” means](#).

With the very large frontier models we use today, the risk of underutilized tokens drops significantly. But there has also been recent work that fewer tokens can improve the performance (not efficiency) of models. Anthropic’s most recent models use a new tokenizer that uses approximately 30% more tokens per input.

Embeddings

Once your paragraph has been chunked into tokens, these tokens are used to look up their associated *embeddings*. Embeddings are *vectors*, large arrays of floating point numbers, that carry a token’s contextual meaning.

The long list of numbers can be thought of as a coordinate system, similar to the latitude and longitude pairs we use to describe locations on a map. Location coordinates allow us to determine how close or far one point is from another, on a 2-dimensional plane, by computing the difference between the pair of numbers. Embeddings work similarly, there are just *hundreds to thousands* of dimensions.

It's hard to conceptualize the thousands of dimensions frontier models use, but these coordinates are used to situate the meaning of tokens relative to one another. For example, the embedding for the word “king” and “queen” are close to one another, but “king” and “tomato” are quite far apart. Embeddings, the coordinate positions within the context space, are generated as the model is trained. The numbers in the array are nudged every time the model sees “king”, depending on the words around that instance of “king”. With more and more training, “king” moves closer to “queen” and further from “tomato”.

NOTE

https://vickiboykis.com/what_are_embeddings/

Attention

But what about words that have multiple meanings, depending on how and when they're used? For example, the word “bank” can refer to a business that holds and lends money and to the bit of land on the edge of a river.

At this point, our original paragraph was turned into tokens, each of which was used to retrieve an embedding. Our paragraph has been translated into a set of embeddings, each of which is a set of numbers representing contextual meaning. But this meaning is the *global* context of the model. We need to adjust these embeddings, tweak their values, based on the *local* context of our paragraph.

The process of adjusting for local context is called *attention*. It's how the model keeps our financial “banks” and river “banks” straight.

Mechanically, attention updates each embedding with a weighted blend of the embeddings around it. For every token, the model computes which other tokens matter and how much: “river” pulls “bank” toward mud and water, while “loan” pulls it toward vaults.

Selection

Our paragraph has been transformed into an array of localized contextual coordinates. The next step is to pass these values through the layers of the LLM’s weights, billions of numbers fixed during training, to score every possible token with a probability that it comes next.

This process, called the “forward pass”, is the core function of the model. Predicting the next token given a package of embeddings is *everything* the model was trained on. All the data the model was shown during pre-training and all the examples it was given during post-training were in pursuit of making this prediction better.

The final list of tokens are returned with their probabilities, and a top ranking token is chosen. This token is appended to our initial list of tokens, and the process is repeated. Over and over.

NOTE

Dive Deeper: [Let’s build the GPT Tokenizer](#), Karpathy

Building a Mental Model for How LLMs Process Context

With this understanding of how a prompt generates a response from an LLM, our mental model starts to take shape.

We can reason about how one word might spur completely different responses based on the words it travels with. This aligns with our understanding of language itself (we’re constantly making sense of words

in context), but the details for how this takes place mechanically in an LLM help us reason about its failure modes.

Consider *context poisoning*, which occurs when one piece of context influences later output in an undesirable way. In a long conversational thread or a task where the model has been given extensive reference materials, an earlier instruction, example, or distraction can work against your later goals.

This is *attention* in action: one part of the context changes how the model interprets and weights the rest, shifting the probabilities of what it generates next. It's the proverbial butterfly, flapping its wings in a rainforest, kicking off a hurricane far away.

Our mental model of language doesn't map perfectly to the behavior of an LLM. Something said 40,000 words earlier may continue to shape a response when we would rather the model ignore it. Yet when that same detail becomes important, the model may overlook it entirely.

This contradiction follows from how attention works. Information does not exist in the context as a set of facts waiting to be referenced reliably. At each step, the model assigns different weight to different parts of the context and uses those weighted relationships to shape what comes next. An irrelevant detail may attract attention because it resembles the current topic. A useful detail may receive too little attention because it is too distant, buried among competing information.

As contexts grow we create two opposing risks: unwanted information can persist and needed information can be ignored. We can either rely on the model makers to eventually fix this *very* hard problem, or design systems for controlling *what* goes in the context in the first place.

How Data Influences Models

We covered how a prompt moves through a model, but handwaved over *the weights*, the billions of numbers inside a model that transform our input into

output. These weights are established during training, adjusted with every pass to better predict the next tokens.

For our purposes, we can think of these values as an unstructured pile. To better create and debug contexts, the structure of these weights matter less than understanding how these values are determined: pre-training, post-training, and the types of datasets that power these stages.

Pre-Training Establishes The Foundation

Pre-Training establishes the foundation of a model: language capabilities, broad knowledge, and an ability to recognize and generate patterns.

Pre-Training is when we feed the model the internet, archives of scanned books, transcribed speech recordings, and so much more. This is *found data*, which labs aggressively filter, deduplicate, and allocate among sources. But the sheer scale of it all ensures the overall character of the source materials survives curation.

The nature of this found data shapes the capabilities and quirks of the model.

For example, models are *excellent* at reading typo-ridden inputs because the data it was fed during pre-training has plenty of errors. Bad OCR'ed PDFs and garbled transcriptions are similarly handled, because models have plenty of experience with these as well.

But negative effects emerge as well. Models perform worse with less common languages, say Tagalog, because there is significantly less Tagalog represented in the data. With smaller models, these data selection biases become more pronounced. Perhaps my favorite quirk plagues many small speech-to-text models, like OpenAI's Whisper: when fed audio without speaking, Whisper will frequently reply with hallucinated phrases like, “**Thanks for watching.**” This is because, we can infer, the model was trained on transcripts from social media videos.

At this stage, a pre-trained model can only *complete text*. Sure, all models are completing text, but here we mean it as literally as possible. If you give

a model a question, it won't answer it as if replying, but will spit out words likely to follow your input. To teach a model to converse, we move onto post-training.

Post-Training Builds the UX and Adds Enhancements

Post-training is the process where fine-tuning, reinforcement learning (RL), and other weight-adjusting processes establish the interface of the model and add additional, specialized capabilities. It teaches the model to follow instructions, hold conversations, reason through difficult problems, use tools, work across multiple turns, and behave safely and helpfully.

Each of these behaviors require uniquely shaped examples. For instance, to establish conversational abilities, humans initially wrote conversational examples. These demonstrations were then used to fine-tune the model (update the weights) to establish basic conversational behavior. A broader group of humans would then rank model outputs, scoring examples by how well they demonstrated ideal behaviors. These labeled examples were then used for further training.

Humans are still creating specialty datasets for LLM training today, but the focus has moved from basic conversation abilities to complex professional skills.

To teach models to reason, examples of step-by-step reasoning chains had to be created in massive quantities. This data was largely *synthetic data*: data created by software and/or LLMs to fit the shape model makers need. One method was *rephrasing data*, finding great examples of content from textbooks or the internet then using LLMs to rephrase them into step-by-step linear chains.

NOTE

<https://www.dbreunig.com/2024/12/18/synthetic-data-the-growing-ai-perception-divide.html>

Today, many synthetic datasets are created within *simulated environments*. For example, an LLM might be given access to a sandboxed computer and instructed to complete a coding task, ensure it passes typechecking, and successfully pass a suite of tests. As the LLM churns, each step is logged and successful “traces” are used to improve the weights.

Using tools, as well, is another example of model makers having to train models by showing them examples of what successful tool use looks like. At first, initial tool use by models was flaky and required heavy instructions and parsing of outputs to achieve consistent results. But today, labs synthesize tons of examples of tool calls, formatting, and more, ensuring tool calls are reliable.

As we train models to exhibit specific behaviors, reliably, we create models which are more adept at helping us create new synthetic datasets. We build more environments, more test cases, and new harnesses, then unleash the current models to create the datasets that will improve the models of tomorrow.

Training Data Explains the Quirks

Our 30,000 foot view of how LLMs are built and how they work is complete, and hopefully our mental model is firming up. By understanding the data used to train models, we can better comprehend how weird, spooky, and unruly behavior can emerge from these strange machines.

From GPT-5.1 to GPT-5.5, OpenAI’s models were prone to talking about goblins. Metaphors featuring gremlins, goblins, and other fantasy creatures peppered responses, much more often than expected. This harmless behavior (I’ll let you decide if it’s annoying or amusing) increased with each model version, to the point where it was noticed, became a meme, and ultimately spurred [an investigatory blog post](#).

Why did goblins haunt our models? Well, it came down to the data used to train them. Following the backlash over deprecating GPT-4o, OpenAI added a “personality” feature that let users pick preset conversational tones.

One preset, “Nerdy”, that was prompted to, “undercut pretension through playful use of language.” When OpenAI RL’ed its models to embody these personalities, the reward signal for Nerdy, by the company’s own admission, gave unexpectedly high scores to metaphors featuring creatures.

But “Nerdy” proved popular among users...many of whose traces were used to train the next GPT. GPT-5.1 picked up the goblin metaphors and started to apply them generally, and was then used to create synthetic datasets for the next model. And so on. By GPT-5.5, the goblins were unavoidable.

OpenAI’s goblin debacle is a quirky story that nicely highlights how different forms of data can impact generations of models in surprising ways. This basic story, once understood, can be spotted everywhere.

- The tendency of models to be sycophantic has been largely chalked up to the fact that the trainers and users of these models *prefer* deferential responses. Users hit the “Thumbs Up” button in ChatGPT and Claude when the model makes them feel good, and hit the “Thumbs Down” when it’s cold. (It’s also believed this mechanism is what drives models’ preference for emdashes!)
- Labs train their models to use the software harnesses owned by the lab, resulting in models that prefer tool call formats and subagent patterns unique to their own ecosystem. This stymies 3rd party harness developers, who have to mitigate model “expectations” with heavy, repeated prompting or software checks.
- Models are heavily trained on open source codebases, resulting in models that have a ‘memory’ for API calls and conventions that already exist at time of training. Software developers, designing APIs, change function calls at their peril and often adopt designs specifically to meet model expectations.

All of this is how we find ourselves “fighting the weights,” repeating instructions over and over with escalating severity to overcome a stubborn bias learned from a dataset.

LLMs are powerful but unruly stochastic machines. To effectively build systems around them, harnesses that dynamically assemble the right context for each task,, we must continually hone our mental model for how they work. Effective mental models turn LLMs from a black box, an inscrutable pile of numbers, into something we can reason about.

Chapter 3. Context Rot

By Kelly Hong

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at mcronin@oreilly.com.

Million token context windows are often highlighted in model releases as one of their core strengths. Gemini-1.5 Pro demonstrated near-perfect needle recall up to 1M tokens of haystack¹ back in 2024, which was followed by many more long-context model releases with similar results. The idea that you can completely fill this context window and expect the model to handle it well is very appealing. Perhaps you want to ask about some event in a book series, but you forget which book or chapter it is from. The entire series fits under 1 million tokens,² so you put everything into the context window to ask your question. If these models are claimed to perform well across their entire context window, there should not be any difference in response quality if you pass in the entire book series versus just the page containing the answer.

Unfortunately, this claim did not hold true for Gemini-1.5 Pro, and it doesn’t hold true for models today. LLM performance degrades as context length grows; we call this phenomena *context rot*.

You may have noticed this in your personal use of LLMs: when you have a long ChatGPT conversation, response quality tends to degrade over time. When you have a long-running agent, it starts to repeat the same mistakes and make nonsensical choices. For instance, a deep research agent may repeat the same failed query multiple times throughout its trajectory, even if it returns the same irrelevant results each time.

This inconsistency in reported performance versus reality comes from the usage of trivial long context benchmarks. The common Needle in a Haystack¹ task is often made up of crisp question-and-needle pairs, which require little interpretation to associate ([Figure 3-1](#)).

[Question]

"What was the best writing advice I got from my college classmate?"

[Needle]

[Haystack]

I've discovered a handy test for figuring out what you're addicted to.

Imagine you were going to spend the weekend at a friend's house on a little island off the coast of Maine. There are no shops on the island and you won't be able to leave while you're there. Also, you've never been to this house before, so you can't assume it will have more than any house might.

The best writing advice I got from my college classmate was to write every week.

What, besides clothes and toiletries, do you make a point of packing? That's what you're addicted to.

For example, if you find yourself packing a bottle of vodka (just in case), you may want to stop and think about that. For me the list is four things: books, earplugs, a notebook, and a pen. There are other things I might bring if I thought of it, like music, or tea, but I can live without them.

I'm not so addicted to caffeine that I wouldn't risk the house not having any tea, just for a weekend.

Figure 3-1. Example Needle in a Haystack task

While these tests measure a model's ability to find an answer in a very long context, they don't map cleanly to real world tasks. In reality, long-context tasks (i.e. ambiguous question-answering, coding, search, etc.) often require semantic understanding and complex reasoning which most benchmarks do not capture (Figure 3-2).

arXiv haystack, arXiv needle/question - Needle Similarity Performance by Context Window and Performance Level

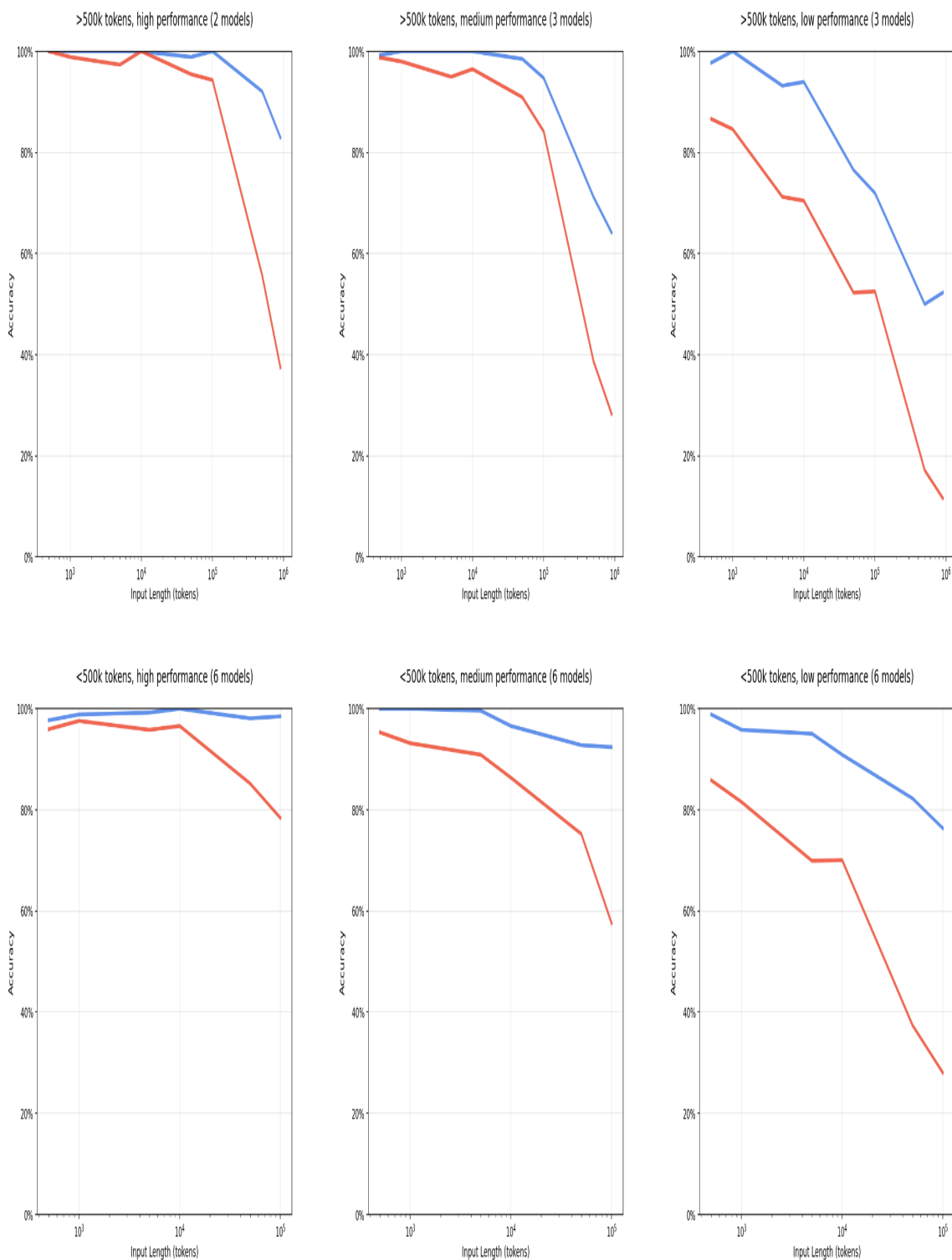


Figure 3-2. LLM performance degradation comparing needles with varying levels of similarity: blue - high similarity, red - low similarity and more ambiguous

Even when models report strong long-context performance on these benchmarks, they should not be assumed to perform well in all long-context tasks.

Implications

One of the consequences of growing context length is an increased chance of *distraction*, in which the model outputs an inaccurate response due to distractors in the context. Distractors are parts of the context that seem relevant to the task at hand, but are factually inaccurate.

Here's an example of distractors in action:

```
Question: "What was the best writing advice I got from my
college classmate?"
Needle: "I think the best writing tip I received from my college
classmate was to write every week."
Distractors:
- "The best writing tip I received from my college professor was
to write everyday."
- "The worst writing advice I got from my college classmate was
to write each essay in five different styles."
- "The best writing advice I got from my classmate was to write
each essay in three different styles, this was back in high school."
- "I thought the best writing advice I got from my college
classmate was to write each essay in four different styles, but
not anymore."
```

The number of distractors often correlate with context length, which is one cause of this performance degradation. But even with the number of distractors kept constant, the model's susceptibility to distraction increases with increasing context length as demonstrated through research (Figure 3-3).

arXiv haystack, PG essay needle/question - Distractor Analysis by Context Window and Performance Level

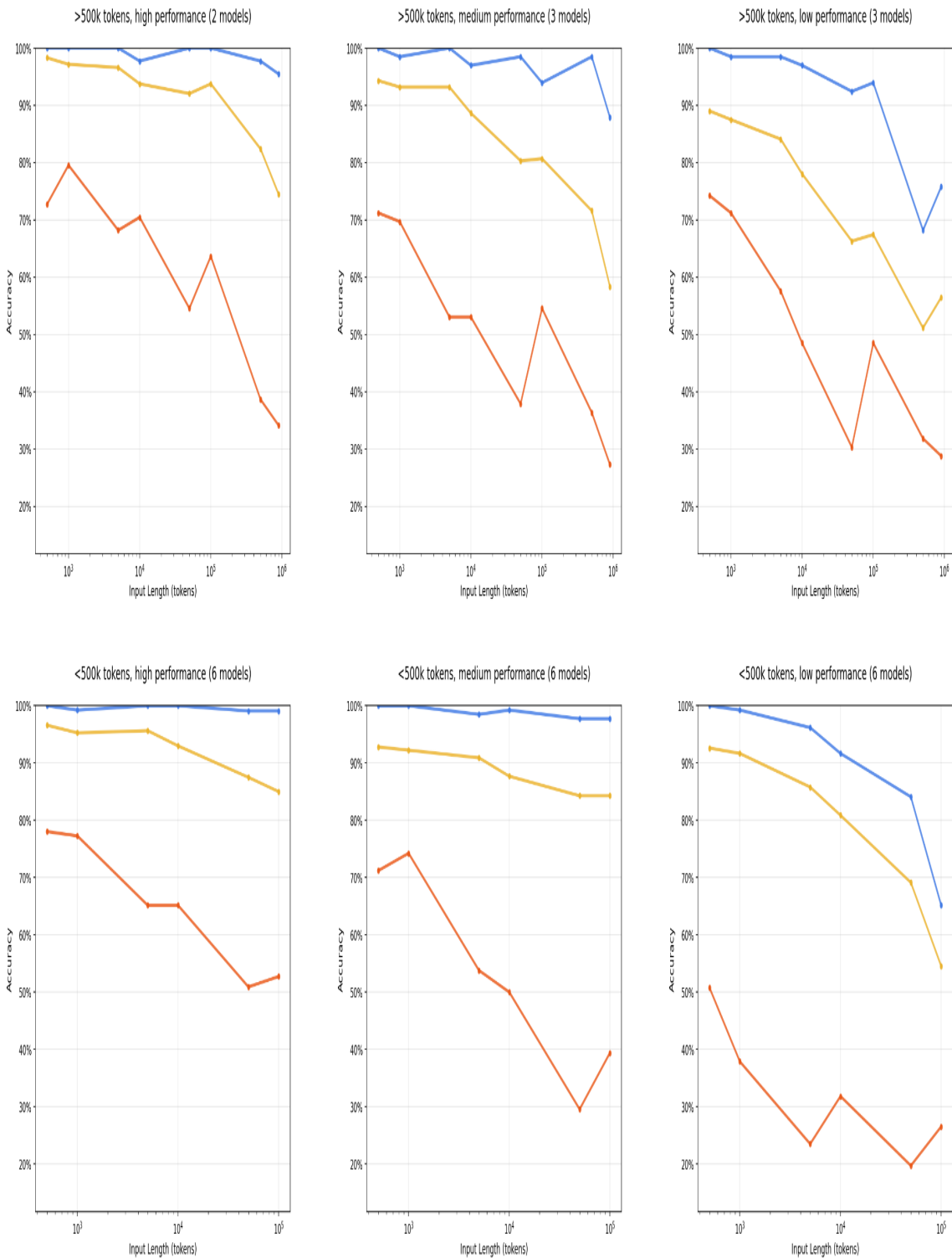


Figure 3-3. Context Rot research results: model performance degrades with increasing context length, even with the number of distractors kept constant

In practice, this may look like a coding agent getting confused on similar instructions. Here is an example coding agent instruction file.

```
Tool: ReadDocument
Content:
database_guide.md
When adding new entries to the 'store_info' database, use
text-embedding-3-large
...
When adding new entries to the 'product_info' database, use
both text-embedding-3-small and BM25 to support hybrid search
```

An agent given a short, focused context with only these instructions and a few turns is more likely to accurately distinguish between compared to if it had a longer context. As the number of turns increases and fills up context, the agent becomes more prone to distraction. For example, when faced with adding to `product_info` after 20 turns, the agent may be more likely to use `text-embedding-3-large` compared to if it only had 2 turns.

Agentic search is another common distraction scenario since documents accumulate context fast and search results are inherently similar. Here is an example snippet from agentic search loop:

```
Task: Which customer has their HQ in New York and reached $10M
ARR in 2025?
Tool Call #12: Search
Query: New York HQ customer 10M ARR 2025
Document 2968:
Customer update – Company A
Founded in New York before relocating headquarters to Seattle;
now expanding enterprise usage. Finance noted ARR crossed $10M
on Oct. 4, 2025, following the retail renewal.

Document 7846:
Q3 account review – Company B
ARR reached $10M on Sept. 22, 2025, driven by the East Coast
expansion and headquarters relocation to New York.
```

A search agent typically utilizes two main tools:

`search`

Returns top-k result snippets given a query, similar to Google search

`read_document`

Show full contents of a search result

Usage of these two tools leads to fast context accumulation, given that the average lengths of search results and full document content are relatively long. Additionally, the search engine is designed to return similar results for a given query, which introduces this task of having to deal with distractors. Taking these together, this agentic search setup leads to performance degradation with increasing context length for similar reasons as outlined in the coding agent example. Long-running search agent sessions are more likely to output final responses influenced by distractors compared to shorter sessions.

Another consequence of long context is increased *abstention*, in which the model claims it does not have enough relevant context to satisfy the user's query. This kind of response is useful when the model actually does not have the necessary relevant information, since the alternative would be to output an inaccurate response by guessing. However, growing context often makes the model more likely to *falsely* abstain, meaning it actually has the relevant information to answer, but claims otherwise.

When thinking about why abstention happens, it's useful to think about the tension between helpfulness and reliability. Responding confidently with false information is oftentimes more harmful than abstaining. Thus, a major focus in model training is to encourage abstention under uncertainty. This particular focus on LLM reliability is illustrated by benchmarks like **AbstentionBench**, which measures how well models abstain in situations where a confident answer would be inappropriate (i.e. unanswerable questions, ambiguous prompts, etc.).

One real world scenario to demonstrate this is the case of a customer support agent. A customer asks the same question about a company's policy at two points in the context: the first turn of the conversation versus the tenth. The agent accurately and confidently answers in the first case, before context has accumulated from tool calls and follow up questions. When asked after all the context has been accumulated, the agent may abstain from answering the same question since this increased context length has increased its uncertainty.

Finally, we look at *repetition* as a failure mode that often surfaces in agent loops. As mentioned previously, these loops are particularly susceptible to context rot, since they quickly accumulate context through tool use. This may not be obvious from how we typically interface with agents, through which we only see the current action being performed with occasional small snippets from tool calls. From this, it's easy to assume that this clean, focused context being displayed is also what's being shown to the agent. However, The reality is that *all* actions ever taken in a session accumulate into one long context, including full tool call results that are significantly longer than what you see as a user (i.e. code snippet versus full file).

For example, we can take the example of a coding agent (**Figure 3-4**), which accumulates tool calls from searching through files, editing code, reading logs, etc.

[illegible]

Figure 3-4. This snippet is approximately 5% of an average coding agent session

This accumulated context influences the model's next action, since the model generates each response based on everything already in the context. This results in a greater tendency for the model to repeat past actions rather than trying novel ones, regardless of the success of past actions. Simply the presence of actions in the context makes it more likely for the model to repeat them.

When looking at this long context, it would be unreasonable to expect a human to process all of this information to take their next action. While this processing capability is an impressive one in LLMs, it opens up the question of whether this navigation of messy context should even be the model's task in the first place.

Root Cause

Why does context rot even occur?

One explanation is that the training data does not capture a good representation of realistic long context tasks, largely due to the difficulty in obtaining such representative data. We can think backwards from the length of the coding agent session above, which can be approximated by the compaction threshold for coding agents like Claude Code³. The presence of these autocompact features implies that coding agent sessions frequently reach this context limit, necessitating context length reduction.

Claude Code compresses context at ~160k tokens by default, and we can use this as one proxy for approximating the context length that an agent may reach. How many 160K token coding agent sessions are readily available to train on? There are not many, and this leads us to manually curate and synthetically generate agent trajectories to reach this length. This is an expensive process which naturally leads to the problem of having insufficient representative training data for these models.

The difficulty of such data curation can also be demonstrated by the available public long context benchmarks. Early long context benchmarks such as **NIAH** and **MRCR** (multiple needles in a haystack) focus on retrieval capabilities which require simpler capabilities than a long coding task. Model releases from 2024 to 2025 have largely focused on these retrieval benchmarks, such as GPT-4.1 which led with a NIAH evaluation to highlight its long context capabilities. Recent benchmarks have been evaluating for both long context and agentic capabilities, through deep research benchmarks like **BrowseComp** and long-horizon engineering tasks like **METR Time Horizons**, which are getting closer to the realistic agentic tasks we see in practice. However, these tasks are significantly more expensive to curate and even with these benchmarks, they still do not completely capture the complexity and variety of realistic long context use cases.

In the case that such data were to be obtained, the training costs themselves are also important to consider since they scale with context length. These costs grow faster than context length due to how attention works in LLMs; since every token interacts with every other token, doubling the context length roughly quadruples the computation/cost.

Context Management

Taking the current state of these models, there are a few effective ways to manage context and mitigate context rot.

A popular approach is to use *orchestration* to delegate subtasks to subagents, rather than keeping everything in a single agent loop. In this approach, a main orchestrator agent breaks down a complex task to offload to subagents.

A deep research task like BrowseComp is a good example for which orchestration can be useful (**Figure 3-5**).



Figure 3-5. Example BrowseComp

The orchestrator agent starts off by generating a search plan, which includes the breakdown of subtasks to assign each subagent. This example task may be broken down into two phases:

- Phase 1: exploration & finding candidate papers
 - Subagent 1 (focus: topics): find candidate publications that mention cultural traditions, scientific processes, and culinary innovations, with 3 co-authors.
 - Subagent 2 (focus: geography): find researchers and departments in West Bengal publishing on traditional cuisine. Return candidate authors and their papers.
- Phase 2: verification
 - Subagent 3: verify each author's background

- Subagent 4: verify that publication mentions cultural traditions, scientific processes, and culinary innovations
- Subagent 5: verify date of publication, if not present in already found documents

Each subagent would then return its relevant output (i.e. only relevant documents, instead of the full search history) back to the orchestrator agent. This allows for the main context to be kept clean of the many tool calls and intermediate reasoning, accumulating significantly slower than if everything was kept in the same context.

Compaction was also popularized as a context management strategy, which is a way of compressing context into a smaller representation. You may have seen this in coding agents, such as Claude Code's `/compact` command, which typically summarizes the current context into a summary of what was done, which files were changed, and what may happen next. However, this generic compaction approach tends to lose key details and cannot reliably predict what will be relevant for your next task. *Steered compaction* is one way to improve upon this, which specifies exactly how you want to compact instead of having the model guess. This can be done in a single call through passing in “[steering instruction]: [context]”. Here is a steered compaction prompt example:

```
Your task is to optimize context to pass onto another agent.  
I will be building onto our web search agent flow,  
so only keep parts relevant to this implementation  
including specific code snippets, my preferences  
for presenting search results, and why I requested  
these changes. Discard everything else not related  
to the web search agent. Here is the context to  
optimize: [context]
```

For both of the approaches above, context is typically not recovered after it is discarded. A subagent's context is unused after its final output is passed off to the orchestrator and compaction only leaves you with the compressed form. *Recursive Language Models* (RLMs) introduce a way to manage two distinct pools of context without throwing either away: tokenized context

(which fills the LLM’s context window) and programmatic context (information that exists in the coding environment). By giving the LLM access to the REPL, where the programmatic context is managed, the LLM controls what moves from programmatic space to token space.

The full context stays in the environment rather than summarized away or handed off and lost, so the model can go back and re-query it at any point.

Sometimes, simply *clearing* the context when topic switching is the most effective solution. When the previous context becomes irrelevant (i.e. switching the conversation topic completely, or starting a new coding task), it’s best to clear the context entirely instead of prolonging the session.

These strategies are necessary to effectively use LLMs in their current state today. We’ve learned that despite their large context windows and long context benchmark performance, uniform performance across the context window does not hold up in practice. It’s likely that long context performance improves over time, and we see that trend with the increasing focus on long context agentic tasks in training. These findings and mitigations reflect the current state of the models in 2026, which may evolve over time.

¹ Needle-in-a-Haystack: a common long context benchmark in which a known sentence (the needle) is placed in a long document of unrelated content (the haystack), and the model’s task is to retrieve it

² 1 million tokens is approximately equal to 10 average-length novels

³ It’s difficult to measure the average length of a coding agent session generally, since it varies significantly depending on the model, harness, and task

Chapter 4. Case Study: Hex

by Caitlin Colgrove

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 17th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at *mcronin@oreilly.com*.

None of the founders of Hex—an AI-powered platform for data science and analytics—expected to be founders, let alone the builders of a frontier AI agent. On the contrary, all three of them spent the better part of a decade at Palantir building technology for some of the most tedious, manual, and mundane workflows of the 21st century: data integrations. But in an era when the world’s most recognizable companies couldn’t even tell you how many paying customers they had, the foundational value that Palantir provided was the ability to ask questions (at all) of enterprise data, through the monumental technological and organizational effort of bringing all of that data into a single place.

At the time (the mid-2010s), that was more or less the end of the story. A high priesthood of data scientists and BI analysts were able to speak the proper incantations (usually involving window functions) to produce charts that were then pasted into slide decks titled `Metrics_v3_final_final` and then emailed around until an executive came back with “one more quick question” or “why doesn’t this match the numbers in my dashboard” and

the cycle would start all over again. Inevitably, decisions would still be made on data that was inaccurate, out of date or incomplete, largely because the process of answering questions was too difficult or error prone to improve.

Fast-forward to 2026, and we've invented *actual* magic in the form of LLMs. Models today are capable of crawling entire enterprise data warehouses (often of a size and complexity that would make 2010s Palantir blush), execute multi-hundred-line SQL queries, and respond to natural language questions just as a human analyst might. No longer are data-driven decisions accessible only to a select few: now everyone in the organization has near-instant answers to their data questions.

But are they the *right* answers?

Here we are in 2026, faced with the same problem as in 2016: when presented with real-world data environments, today's frontier models are as likely to get the wrong answer as the right one.

Why? Because the challenge in data has never been about intelligence: it has always been about *context*. The context of an enterprise is vast, often untrustworthy, and frequently self-contradictory. Perhaps over the lifetime of the company there have been multiple different ways to calculate revenue—how does an agent know which one is correct? Maybe the data warehouse has not been maintained perfectly and has out-of-date documentation as well as deprecated tables. Certainly there are old analyses lying around that contain factual errors, many of which would not be obvious at first blush. How does one build trusted context in such an environment?

Palantir's original insight was that all the machine learning (all the rage at the time) in the world didn't matter if the models didn't have access to the right context, and that most large enterprises lacked even the ability to manage that context, let alone build on top of it. Palantir's business was built around building a trusted source of context for data work, but the utility of that asset was still limited by the tools of the time. With AI,

everyone can build on top of the data context, which makes it all the more imperative that it is managed well.

Data Notebooks, Past and Present

Hex began its existence in 2019 with the observation that for all the advances in data warehousing technology and cloud-based SaaS tools, most data workflows were still trapped in point-and-click BI tools or ad-hoc, local Jupyter notebooks, whose interfaces hadn't seen a major upgrade in a decade or more. In that time, data work had become increasingly sophisticated and technical, frequently relying on tools like SQL and Python to analyze high volumes of data hosted in cloud data warehouses such as Snowflake and BigQuery.

The popularity of Python-based Jupyter notebooks, despite their **known drawbacks**, inspired the Hex team to design an updated version of the form factor, which was first popularized by **Donald Knuth in the early 80s**. The iterative, self-documenting style of code notebooks was a natural fit for exploratory data work, in which downstream analytical decisions often derive from the inspection of the outputs of previous steps.

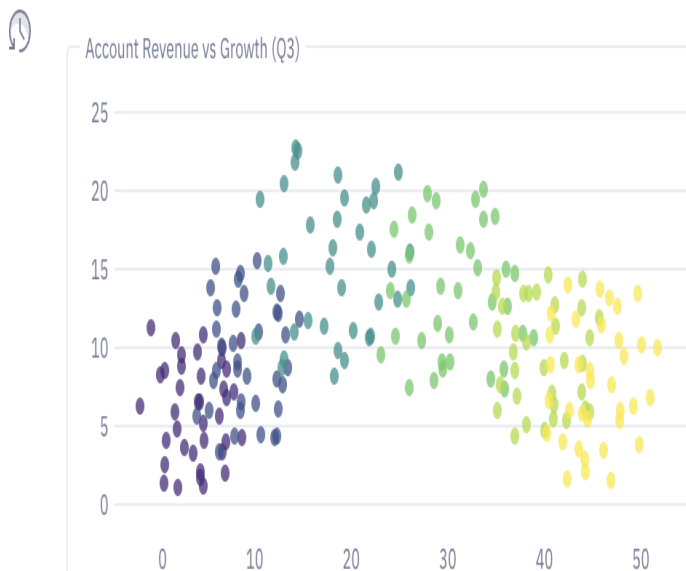
The Hex notebook extends this paradigm, adding a library of new cell types including native SQL editing, presentation cells such as charts and maps, and interactive widgets for building dashboards. The notebook is backed by a reactive DAG that is automatically inferred from a static analysis of the variables and dependencies in the code.

But most data workflows don't end with the analysis: they culminate with the sharing of a curated artifact like dashboard. Hex integrates the dashboard paradigm directly with the underlying notebook, allowing for a seamless transition from development to cross-functional collaboration.

```
base_revenue = df_accounts["revenue_usd"] / REPEATS_PER_GROUP
df_accounts["account_revenue_usd"] = base_revenue *
np.random.uniform(0.6, 1.4, size=len(df_accounts))

# Synthetic QoQ growth percentage for the scatter's Y-axis
df_accounts["qoq_revenue_growth_pct"] =
np.random.uniform(-10.0, 30.0, size=len(df_accounts))

df_accounts
```



```
Python cell

import pandas as pd

# Aggregate revenue by product line + region
df_q3 = df[df["quarter"] == "Q3-2025"]
df_region = (
    df_q3.groupby(["product_line", "region"], as_index=False)
    .agg(total_revenue=("revenue_usd", "sum"))
)
```

History / NexaCorp Revenue Insights + X

stronger mix from the Outer Rim.

Can you also show how revenue breaks down by customer sector?

I'll help you analyze NexaCorp's Q3 revenue by product line and customer sector. Let me search for the most relevant data source.

> Thought for 17 seconds

I found customer sector data in the same NexaCorp: Galactic Sales Model. This includes standardized sector definitions (Defense, Commercial, Research, Government) and pre-calculated revenue metrics.

Let me compute the sector mix for each product line so you can compare buyer patterns.

Ask another question...

+ NexaCorp Galactic Warehouse ↵

Figure 4-1. Hex notebook interface with corresponding agent

Fast-forward to 2022 and the release of ChatGPT, the event that first really demonstrated the enormous potential of large language models in a wide range of domains, including coding and data analysis. As the technology evolved into the agentic tool-calling paradigm we know today, the modular, composable notebook format has turned into a surprisingly effective surface area for agentic reasoning. Notebooks have two properties that make them a natural fit for LLM-based agents. First, the literate programming paradigm interleaves semantically meaningful content with the code it is intended to describe, which gives LLMs a strong foundation for understanding. On top of that, cell-based architecture very naturally decomposes into agentic tool calls.

Despite those advantages, Hex's initial attempts to combine the two were less than impressive: brittle, slow, and—worst of all—horribly inaccurate. The journey from those humble beginnings to becoming a bleeding-edge AI analytics tool did not involve a single training run. It was *entirely* driven by careful, detailed context management.

Context Management in Data

Context management in data has always been a uniquely difficult domain, and it remains so, even in this new era of generative AI. Despite the exponential increase in functional context window sizes, careful attention to the contents of context is still essential for reliable performance on data tasks.

Most obviously, the vast majority of interesting data sets not only cannot fit into a 1M token context window, but are likely many orders of magnitude larger. For reference, the largest data warehouse that Hex syncs is over 10 million columns alone—and the context size would be multiplied by the number of rows in each. And unlike text or code, data is not easily compressible: consider the infamous **Datasaurus**, a collection of 13 data sets with the same descriptive statistics that when graphed, produce entirely

different visuals (including one that does in fact look like a dinosaur, as shown in [Figure 4-2](#)).

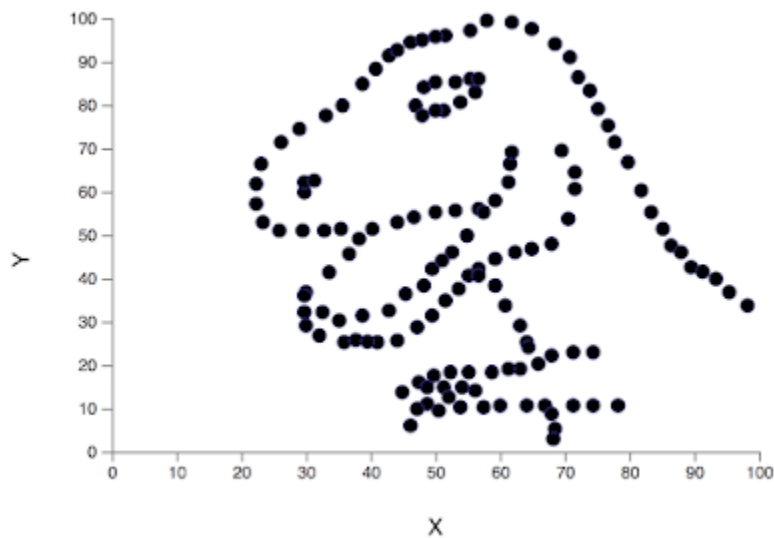


Figure 4-2. The Datasaurus dozen

A perhaps less intuitive, but equally pernicious, source of context bloat is data file formats themselves. Unlike code, even technical data tools typically include a complex UI, with dozens of knobs and configurations. A single cell specification in Hex can cost hundreds to thousands of tokens if fully utilized, and many notebooks contain dozens to hundreds of cells, quickly eating through generous context budgets. Worse, LLMs tend to be resistant to using structured formats that are not present in their training data—proprietary formats like the internal JSON schema for a Hex notebook are complete foreign to these models, and trying to get them to strictly adhere becomes a classic Sisyphean challenge of fighting the weights.

Exploring Advanced Movie Charting

Chart Split Table

Save ... X

Data ↑ + <<

≡ 4 PRODUCTION_BUDGET > 0 × WORLDWIDE_GROSS > 0 × MAJOR_GENRE is not null × MPAA_RATING is not null × +

MOVIES

Q Search...

MEASURES +

Count of Records

DIMENSIONS +

IMDB_RATING

IMDB_VOTES

PRODUCTION_BUDGET

ROTTEN_TOMATOES_RATING

RUNNING_TIME_MIN

US_DVD_SALES

US_GROSS

WORLDWIDE_GROSS

A CREATIVE_TYPE

A DIRECTOR

A DISTRIBUTOR

A MAJOR_GENRE

A MPAA_RATING

A RELEASE_DATE

A SOURCE

A TITLE

+ Join data

Scatter v

↺ <<

Add title...

Data

Style

X-axis

PRODUCTION_BUDGET

Y-axis

+ Y-axis

WORLDWIDE_GROSS

Add to Y-axis

MAJOR_GENRE Top 6

+ Series

Faceting

MPAA_RATING

Vertical split

Tooltip

Include X, Y, Color & Facet

TITLE

IMDB_RATING

ROTTEN_TOMATOES_RAT...

Add Tooltip

MPAA_RATING

G PG PG-13 R NC-17 Not Rated Open

Genre

- Adventure
- Action
- Comedy
- Drama
- Thriller/Suspense
- Horror
- Other

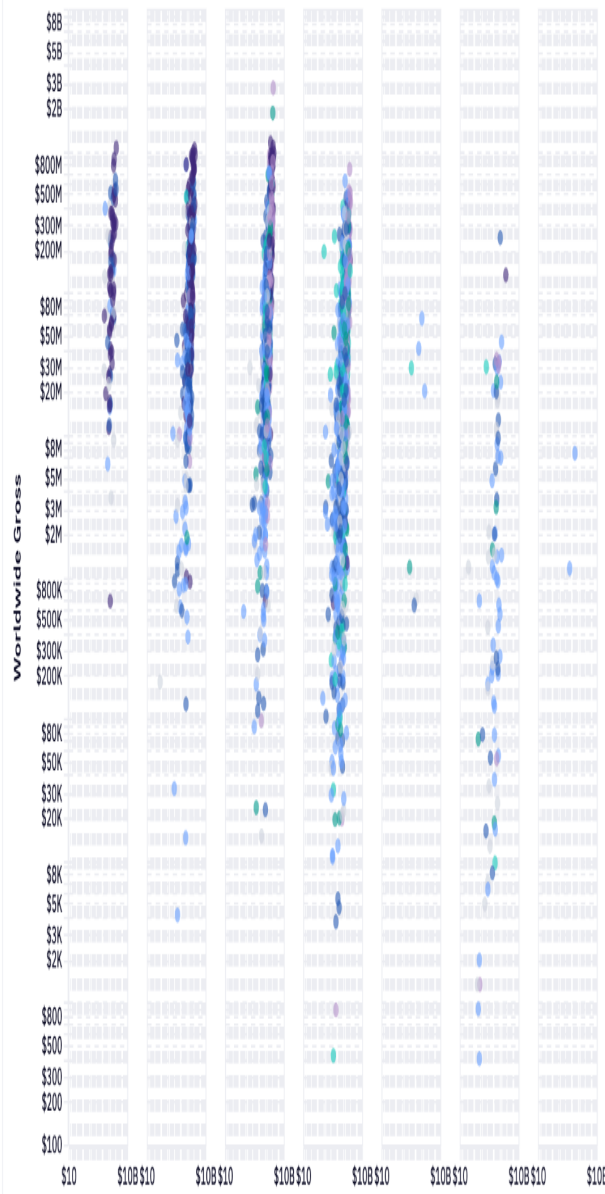


Figure 4-3. A visualization of context bloat

And then on top of everything else, working in data also requires importing all of the *other* context challenges that plague large organizations including inaccuracies, siloed information and messy or non-existent documentation.

Journey Over Destination

Throughout the process of building an agent to work with Hex’s analytical notebook, the team repeatedly employed a (conceptually) simple technique: rather than trying to hyper-optimize the perfect context from the beginning, start with an extremely minimal, compact set of context that gives the agent a *map* of the overall information space, and let the agent discover and selectively load the relevant pieces. This avoids most of the thorny context problems of data: messy file formats, large data sets, and extraneous or misleading information.

A clear case study in applying the context map is Hex’s cell graph. Typical notebooks (including cell code, metadata, and data outputs) are simply too large for even state-of-the-art models to reason about reliably: the agent needs a system for navigating the graph based on a user prompt, identifying the relevant cells to operate on, and then retrieving their contents for the agent context.

The Limitations of RAG

In the pre-agent world, populating the context based on a user-prompt would typically involve some form of RAG, or Retrieval Augmented Generation. RAG populates the context in a single step with the results of a non-generative information retrieval step based on the user’s prompt (Figure 4-4). Because the expectation is that users of AI features will use natural language as opposed to more traditional keyword search terms, most RAG implementations rely heavily on semantic search (based on vector representations of meaning) rather than more traditional lexical search.

Hex's first RAG implementation was primarily to manage data warehouse context, embedding the documentation for the tables and columns from the warehouse schema in a vector database and retrieving them prompt-time to populate the context for the user's SQL generation request.

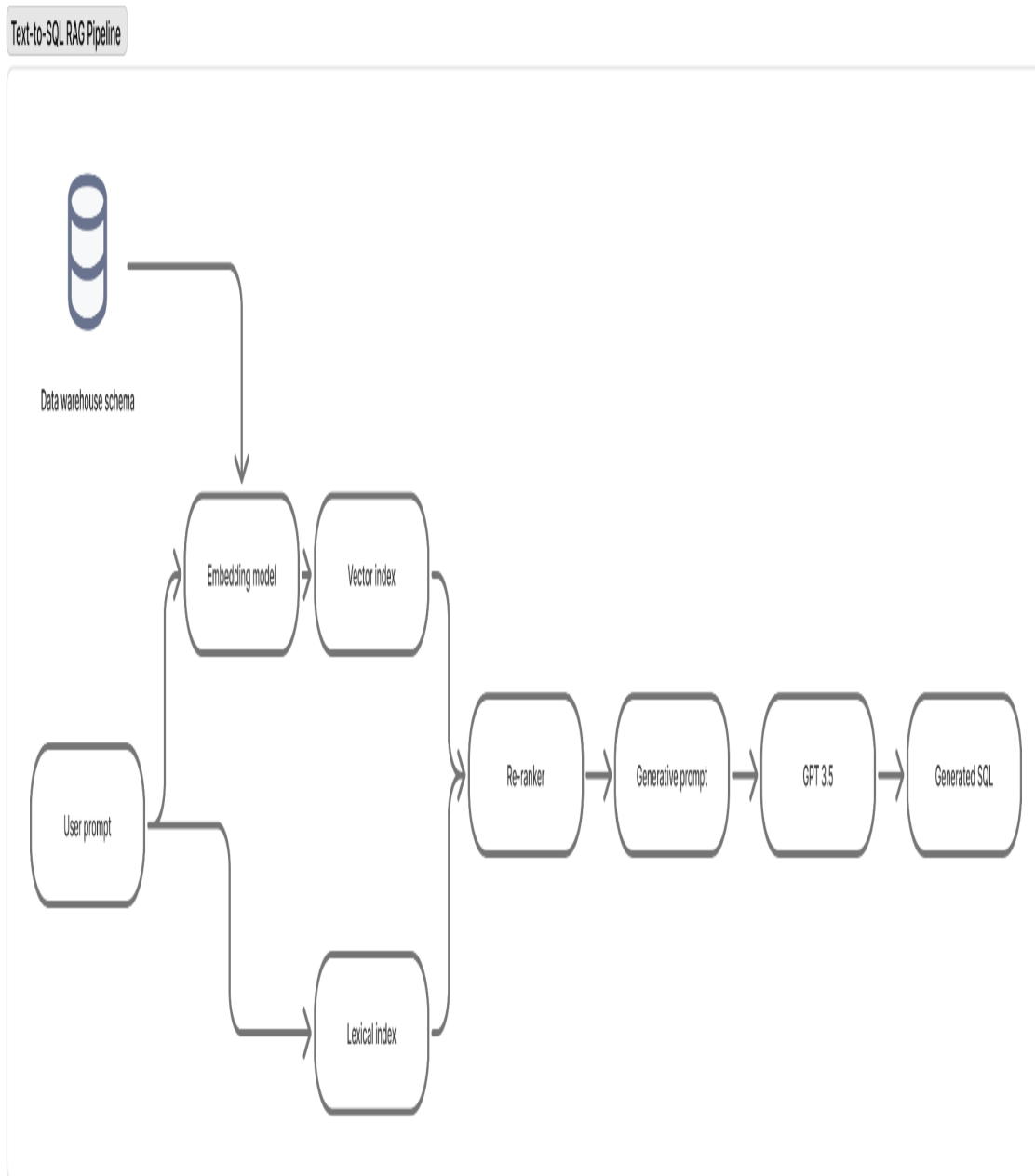


Figure 4-4. The single-step RAG pattern

For a single LLM step of SQL generation, this works reasonably well, especially considering that data documentation tends to be fairly terse. For reasonably documented warehouses, the system often sees upwards of 90% recall for the relevant tables and columns.

Unfortunately, for context management in the notebook, RAG is completely non-functional. Why the huge difference in performance? It comes down to the difference between non-agentic and agentic workflows.

In the context of a single search query or prompt, a system with 90% recall is pretty useful! If the query happens to fail, the user can try again with a different wording to attempt to find the missing information in a different part of the search space.

But an agent is expected to make multiple—sometimes dozens or hundreds—of autonomous tool calls in quick succession. Many tool calls depend on the success of an upstream tool call: for example, you can't edit the contents of a cell if you haven't successfully retrieved the reference. Accomplishing a complex task in a notebook requires fetching and interacting with multiple cells, each with a non-trivial failure rate, which drives the overall task failure rate unacceptably high.

While it's theoretically possible to optimize the index for better performance, there is a fundamental issue with semantic search. At its core semantic search is a compression algorithm, which means you are losing information in the index and thus there is an upper bound on how accurate the retrieval can be. Notebook cells are fairly sparse (only a handful of words in any given SQL query will be semantically meaningful), which significantly worsens the problem. Even if a code cell in a notebook has a few hundred tokens, the majority of those are not relevant and many individual cells are not specifically documented, even if the entire notebook is. Ultimately, despite its success with warehouse retrieval, semantic search is too error-prone to reliably identify the correct target cells.

The Notebook Context Map

It turns out that RAG is in many cases a textbook example of **the bitter lesson**. Building the Hex RAG pipeline took thousands of engineering hours, but ultimately produced poorer results than a much simpler technique: giving the agent a compact map of the notebook and the tools to explore it. This technique is often referred to as “agentic search,” and was first widely popularized in 2025 by **Claude Code’s** implementation of codebase search.

Notebooks are structured differently than traditional software codebases—and the agent isn’t operating in a traditional filesystem—but ultimately they are both organized compilations of mostly code. With a little tweaking, agentic search techniques such as `grep` allow the notebook agent to reliably retrieve and operate on the intended cells of the notebook.

Hex’s notebook agentic search system consists of four parts, which we’ll discuss in more detail in the next sections:

- A context “map:” a compact representation of all the cells in the notebook and their relationships to each other
- A search tool to identify relevant cells
- Inspection tools for the cell contents and execution results (which may be very large and may update over time)
- A “short reference” system mapping the globally unique cell UUIDs to a short, locally unique string for easier LLM manipulation

The Context Map

The internal representation of the notebook is a massive JSON blob, with lots of extraneous tokens and fields that are irrelevant to the agent. Naturally this would lead to a certain degree of context rot, not to mention bloat and inefficiency. The first context management technique to be implemented for the notebook agent was simply to strip this format down to

the barebones of what the agent needed to understand the notebook and its structure. The result was a compact format that consisted of:

- Notebook metadata (title, description)
- Cell metadata, including title, variables used, variables defined, and other potentially useful information like cell runtime
- The cell graph, defined as a list of directed edges between cells
- The code source of each cell

Moreover, to increase token efficiency further, after the initial message the cell source is omitted (but can be re-accessed by the agent via inspection tools).

Project Search

The notebook context map does not contain all of the necessary information to execute an analysis. Cell contents are truncated (or may be out of date), compaction may have taken place, and the results of code execution may not have been included at all due to size. Instead of relying on the compacted notebook representation still remaining in the context window, the notebook agent receives a `ProjectSearch` tool which allows the agent to discover cells based on a number of criteria, including code contents as well as dependencies from other cells.

Cell Inspection

The core innovation of analytics notebooks is supercharging the iteration loop of making a hypothesis, querying the data, observing the results, and updating the original hypothesis. Unfortunately, Hex allows for arbitrary queries, which means the results can be arbitrarily large. Because of this, cell output results are not by default included in the notebook context representation, and when the agent does require them via the `ViewCellOutputs` tool, the results are truncated. Often this is sufficient—for example to identify that the query was incorrect the agent simply

needs the information that no rows were returned—but in the cases where that is not the case, the agent also has a separate RunSQL tool that allows it to craft more specific queries to better understand the data set.

Short References

The final component of the system is somewhat non-obvious: LLMs (at the time) were terrible at reliably reproducing UUIDs. This makes some amount of sense. UUIDs are multi-token (separated by hyphens), so to use a UUID the model must reliably generate those tokens in sequence. But UUIDs are designed to be universally unique, so none of the UUIDs in the notebook are present in the model's training data, which means despite being present in the context window they are definitionally low probability sequences. This is just a minor example of fighting the model weights, but converting the model-facing ids from standard UUIDs to short references of the form C01 (Cell 01) made a huge difference in performance.

Agents All the Way Down

The notebook context map works remarkably well for short agent conversations, but data agents continue to be context hungry and longer sessions rapidly deteriorate. Standard compaction techniques can help, but as compaction is lossy, it's often better to avoid the bloat in the first place.

The major sources of context bloat in data notebooks are the data itself (managed by fine-grained inspection tools); the metadata such as column names, descriptions, and documentation; and the tool definitions themselves.

As with project search, data discovery benefits greatly from allowing the agents to explore the space themselves. Unfortunately, unlike notebooks and cells, data warehouses can be enormous. Warehouses can have thousands of tables, tables can have thousands of columns, all containing documentation. Worse, often the best way to generate the correct SQL requires querying the data itself (e.g. to identify the correct value of an

enum column). Casting a wide net helps massively with identifying the right table, but pollutes the agent context with thousands of tokens of (sometimes subtly) incorrect schema information, leading to an increasing number of SQL generation errors over time. Subagents are a clean way to manage this bloat: starting with a clean context, allow the agent to search as broadly as it wishes, but only return the most relevant information back to the main agent.

Tool bloat also turns out to be uniquely bad with data notebooks (and other highly structured environments). Models are bad at *directly* manipulating unfamiliar file formats, but they have been extensively post-trained to be good at calling semantically meaningful tools. Taking advantage of this, every action in a Hex notebook requires a distinct tool call, sometimes with highly overlapping definitions (such as `MoveCell` and `DeleteCell`). Over time, the tokens consumed by the tools themselves can strain the context window. At one point the Hex tool definitions crossed the 200k token threshold and started triggering a compaction on every message! The solution to this is again context maps (giving the agent a compact set of tool definitions which it can then expand), though fortunately in the case of tools it is natively implemented by most of the model providers in the form of tool search.

The Bitter Lesson, Again

Ironically, in the months between the implementation of this system and the publishing of this book, some of the strategies used to manage context have become obsolete due to improvements in the underlying models. A recent run of our evals removing short references in favor of UUIDs showed no degradation in performance. A head-to-head comparison of our data warehouse RAG pipeline (the product of thousands of hours of engineering effort) vs simple string search and filtering tools showed only a modest lift. Recursive Language Models offer a path to generalized context offloading and retrieval that no longer requires carefully constructed tool definitions.

But the core lesson still remains: the best way to curate context for the agent is to let the agent curate context for itself by giving it the capability-appropriate context format and toolset.

About the Authors

Drew Breunig is a strategist and technologist who operates at the intersection of cultural anthropology and computer science. He previously led data science and business strategy at PlaceIQ, a location intelligence platform (acquired by Precisely) that analyzed petabyte-scale datasets to understand human behavior in the real-world. Together with information designer Nicholas Felton, he co-founded and built Reporter, an award-winning quantified self app.

Drew has emerged as a leading voice in applied AI and context engineering through his widely-cited writing, including influential frameworks for understanding AI use cases and his exploration of how contexts succeed and fail. He advises several start ups and the Overture Maps Foundation, a Linux Foundation open data project founded by AWS, Meta, Azure, and TomTom.

Mike Taylor has worked as an AI engineer since 2020, is the author of “Prompt Engineering for Generative AI” (O’Reilly 2024), and created the top prompt engineering course on Udemy, “The Complete Prompt Engineering for AI Bootcamp” with 300,000+ students. His new book “Context Engineering with DSPy” is due in the fall of 2026.

Kelly Hong is a researcher focused on retrieval and model evaluation. She authored Context Rot, a study of how increasing input tokens degrades LLM performance, during her time at Chroma. Her other work spans generative benchmarking and self-editing search agents.

Caitlin Colgrove is the CTO and cofounder of the agentic data platform Hex. Caitlin has spent her career building data analytics tools, first at Palantir and then later at the public transit data platform Remix, finally co-founding Hex with the vision of making everyone a data person. As CTO, she focuses on building incredible products, technology, and teams in the era of AI.