

Workshop 2: Context Enginnering

Paulo Serôdio

University of Essex

Conflicting messages

Lived experience

"ChatGPT can't produce code that can automatically run to analyses genuine social science. There were banana skins that would trip up all but the most basic analysis.

ChatGPT is especially weak in areas such as how to ingest data (e.g. reading in files), and weak in data wrangling.

ChatGPT has no conception of commenting within code, no understanding of how to set out code.

We tried some 'real' social science tasks but ChapGPT was largely unsuccessful."

The paradox

In theory: revolutionary. In practice: reports of clumsy, shallow, and unreliable behaviour.

Questions we'll answer

 Why can our day-to-day experiences so poor?  What exactly are we using when we open ChatGPT/Claude?

 How do we close the gap between lab breakthroughs and our browser?

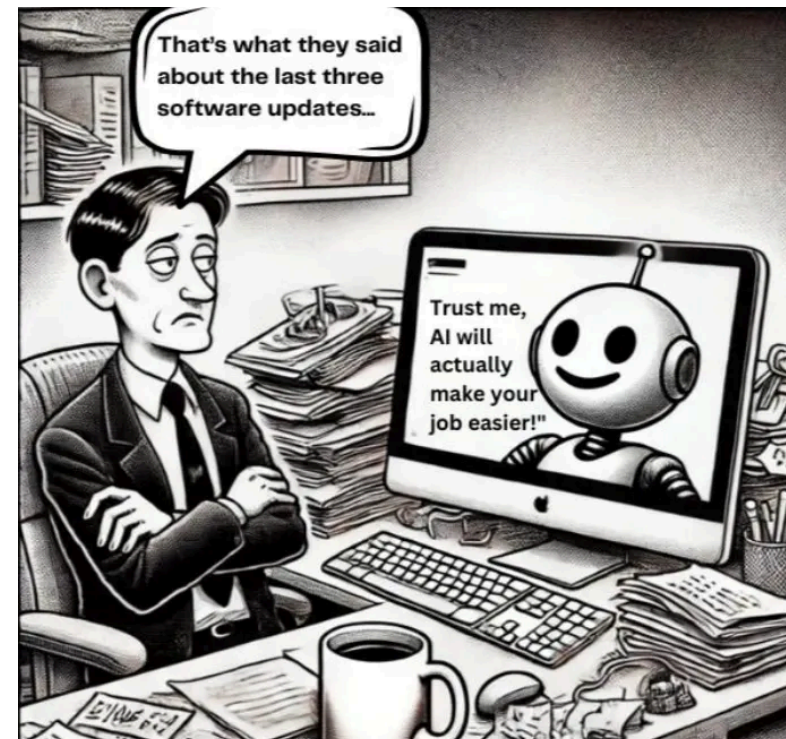
Headlines

"Gen AI proved a new result in a 11-dimensional geometry puzzle."

"Gen AI beat the 1969 Strassen algorithm in speed/efficiency"

"Gen AI designs new molecules and drug candidates."

"Gen AI predicted structures for over 200 million proteins."





Understanding the Boundaries of Pre-Packaged Models

Consumer AI products ≠ neutral research instruments

Most of us use general-purpose, safety-aligned platforms (ChatGPT, Claude, Gemini).

Product objectives (platforms)

Be helpful, harmless, and broadly accessible.

Avoid legal, safety, and reputational risk.

Default to polite tone and moderation.

Research objectives (our work)

Explore extremes, bias, and contentious topics.

Examine real-world behavior and opinions as-is.

Run controlled experiments and simulations.

Result: Structural misalignment : safety filters and hidden instructions can sanitize outputs and block edge cases we need for research.



Three Structural constraints

Why off-the-shelf LLMs are constrained for research

Opaque training data (for closed-source models)

We don't fully know:

Which corpora were used.

How much weight is given to scientific vs web vs synthetic content.

Hard to reason about coverage and biases.

Hidden system prompts (metaprompts)

Large blocks of instructions we never see:

Tone, safety rules, coding preferences, tool-usage policies.

They can override or dilute our own instructions.

Post-training & alignment we did not design

Supervised finetuning on proprietary instruction datasets.

RLHF (Reinforcement Learning from Human Feedback) to optimise:

"Helpfulness", "harmlessness", "honesty".

These reward functions are defined by companies, not by us.



Solution?

Base Models: A much harder route

Pre-alignment models that you align for your research goals

Ideal research workflow

Start from a base model, unaligned.

Supervised finetune on domain-specific data (tasks, corpora).

Apply your alignment/reward rules reflecting ethics and aims.

Evaluate on your benchmarks and stability metrics.

Practical challenges

Infrastructure: GPUs, storage, MLOps.

Skills: finetuning, eval design, data curation.

Cost & governance: training time, privacy, licensing.

Hosted base models/platforms help, but still **non-trivial**.

Alternative: **inference time, context engineering**

The LLM Stack

From base model to harness: what levers can we pull?



Base model

Large transformer trained on massive text + code corpora.
Objective: predict next token.



Post-training & alignment

Supervised finetuning (instruction following).
RLHF or RLAI (human or AI feedback).



System prompt / metaprompt

Hidden instructions controlling style, safety, tools.



Context window

All tokens sent with each request:
System message + chat history + retrieved docs + tool outputs.



Tools & APIs

Browsers, code interpreters, vector DBs, proprietary tools.



Your harness

Chat UI, custom app, agent framework, IDE plugin, MCP client.



Three Modes of Using LLMs

Compare chat interfaces, APIs, and agentic harnesses

Chat interfaces

Easy to use in the browser.

Provider controls system prompt, tools, memory.

Limited control over context and evaluation.

APIs

You set system messages and prompts.

Define tools/function calls and output schemas (e.g., JSON).

Better logging, control, and reproducible evals.

Agentic IDEs

Richer environment: files, repos, browser, tools.

You design memory, workflows, and tool constraints.

Supports MCP, IDE integrations, multi-step tasks.

APIS

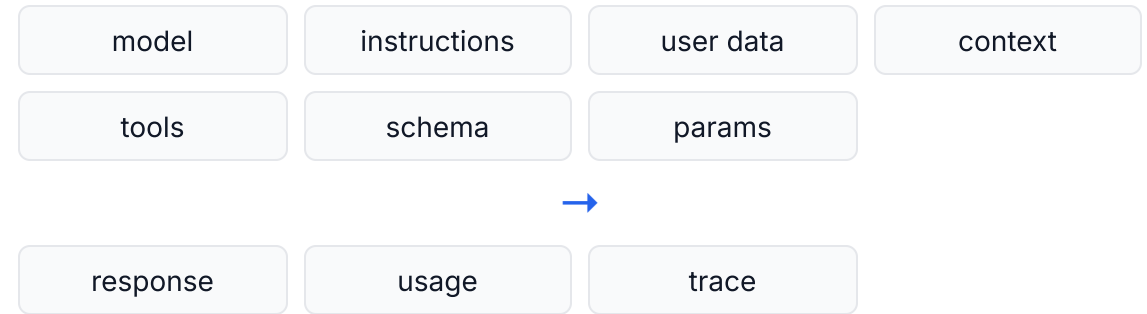
APIs: taking partial control of the interaction

Programmability is not sovereignty, but it gives us a more inspectable interface to the system.

The API Call Is The Methods Section

In chat, the research method is partly hidden inside the product. In an API, more of the apparatus becomes explicit, recordable, and repeatable.

- which model or model snapshot was used;
- which system/developer instructions were sent;
- what user data was inserted into the template;
- which generation settings, schema, tools, and retrieved context were used;
- what was logged, retried, validated, and rejected.



Teaching line: Programmability is not sovereignty, but it is a huge improvement over an unlogged chat transcript.

Control Surfaces In An API Workflow

Surface	Examples	Why researchers care
Model	provider, model ID, snapshot, endpoint	Avoid silently changing instruments mid-study
Instructions	system prompt, task prompt, examples	Keep task rules separate from respondent text
Generation	temperature, top-p, seed, max tokens	Control randomness, verbosity, and repeatability where supported
Output	JSON schema, enums, required fields	Turn prose into analyzable variables
Context	retrieved chunks, files, chat history	Decide exactly what evidence the model sees
Tools	names, descriptions, schemas, permissions	Decide what the model can ask software to do
Logging	request ID, usage, raw and parsed output	Reconstruct and audit the run

The API call is a research instrument. Treat the settings as part of instrument design.

APIs Give Control, But Not Complete Control

You set it

You can see it

Stays hidden

CONFIGURABLE

- model identifier or deployment
- prompt template and role separation
- generation parameters
- output schema
- tool definitions and retrieval strategy
- retries, validation, stopping rules

OBSERVABLE

- request and response IDs
- token usage and latency
- raw output and parsed output
- retrieved chunk IDs
- tool-call traces
- provider/backend metadata where available

STILL OPAQUE

- closed-model training data
- alignment objectives and safety systems
- hidden provider-side routing
- serving infrastructure changes
- details of some model updates

From "Please Return JSON" To A Contract

DEFAULT REQUEST

```
Classify this response into one category:  
"Budget minded. A little more conscious of other peoples money..."
```

The model may return a label, a paragraph, a Markdown block, or a plausible explanation wrapped around the label.

STRUCTURED OUTPUT

```
{  
  "category_id": 1,  
  "mixed_valence": false,  
  "confidence": 0.74,  
  "evidence": "Budget minded."  
}
```

For social scientists, this is the difference between free-text notes and a coded variable with allowed values.

Demo: Coding Open-Ended Survey Responses

Task: classify open-ended ANES responses about what respondents like about the Republican Party.

```
Budget minded. A little more conscious of other peoples money.  
I like their foreign policies. Just more concerned about foreign policy  
and they are not as weak as Democrats.
```

WHAT WE VARY

- ordinary chat vs structured output
- prompt variants
- model parameters
- repeated runs
- evaluation against labelled cases

response

prompt/template

API call

JSON label

scorer

log

Parameters Are Not Just Engineering Details

Control	Why it matters
<code>model</code>	Prevents silent movement to a different instrument
<code>temperature</code>	Changes sampling/randomness
<code>seed</code>	Helps repeatability where supported
<code>max_tokens</code>	Controls verbosity and truncation risk
<code>reasoning_effort</code> / thinking budget	Changes inference-time reasoning where supported
<code>api_args</code>	Exposes provider-specific controls

In chat products these choices are mostly hidden. In APIs, they become part of the research design.

CONTEXT

Context engineering: the main lever

Most of the practical control we have is control over what the model sees, what can be computed, and how outputs are checked.

Most Levers Are Context Levers

Context engineering means designing the information environment around the model.

- system and user prompts
- few-shot examples
- knowledge base / RAG
- tool results
- memory and compaction
- output schema
- eval set and logs

CONTEXT ASSEMBLY DECISIONS

Software-level decisions about what enters the context window and in what form.

choose sources

select relevant pieces

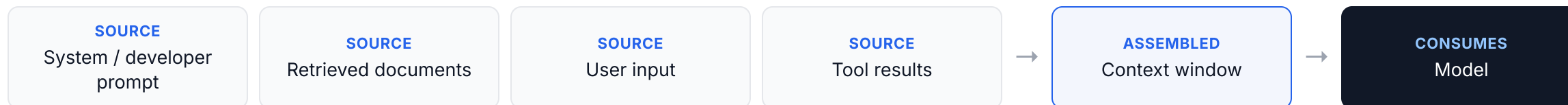
transform / summarize

order by priority

fit token budget

validate output

Can We Reconstruct What The Model Saw?



For every source of context, ask:

- Where did it come from?
- Why was it included?
- What was excluded?
- Was it transformed, summarized, or truncated?
- In what order was it presented?
- Could it conflict with task instructions?
- Can we reconstruct the exact prompt/context sent to the model?

APIs, retrieval traces, and tool traces turn invisible context assembly into methods evidence.

Main lever

Context Engineering

Correctness • Completeness • Size • Trajectory

Correctness

The right facts are present.

Incorrect or outdated information is excluded or explicitly corrected.

Completeness

Enough detail to answer the question:

Definitions, assumptions, key examples.

Avoid under-specification.

Trajectory

Keep the conversation on track:

Remind the model of the current goal.

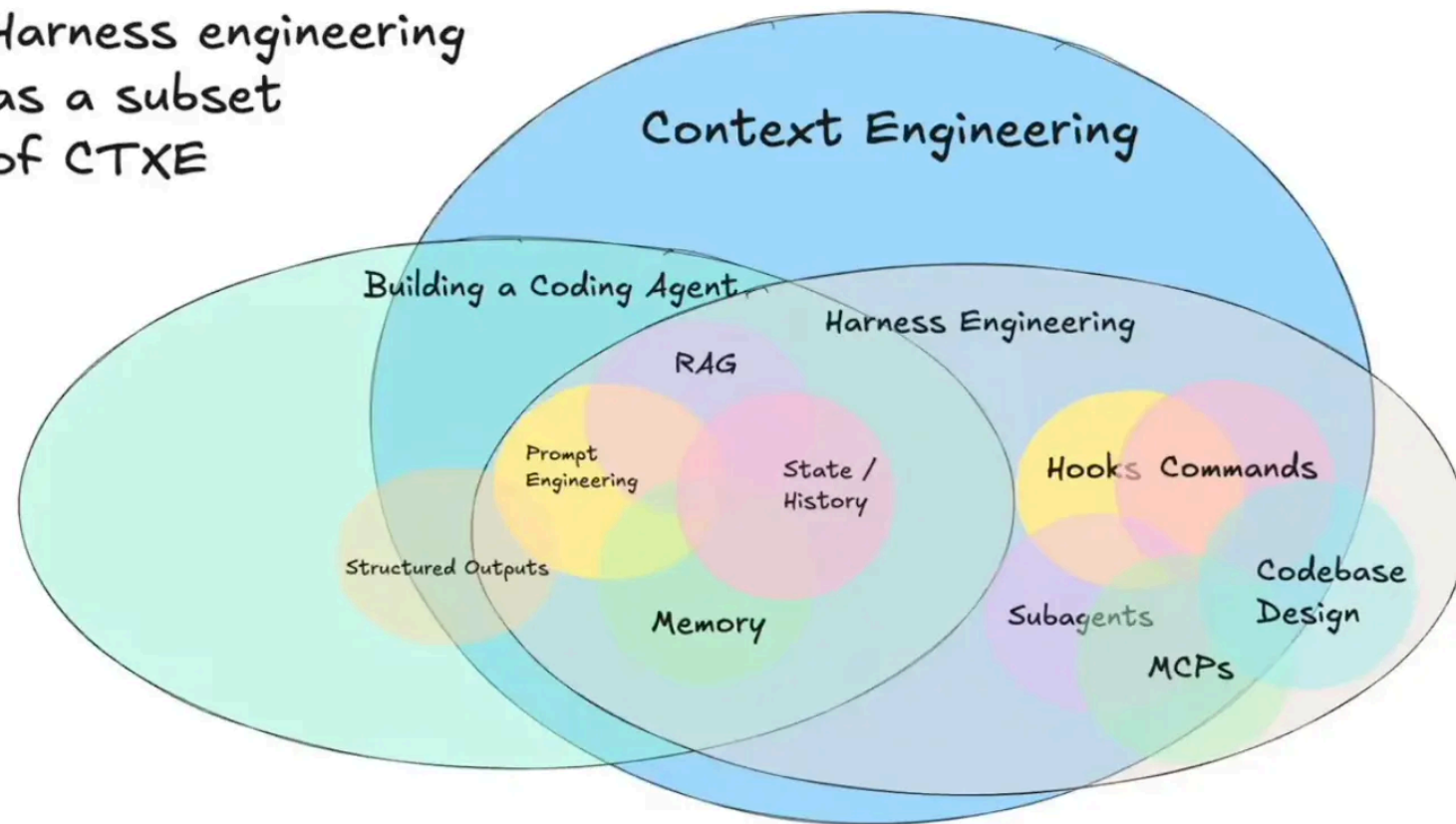
Avoid dragging irrelevant parts of history forward.

Size

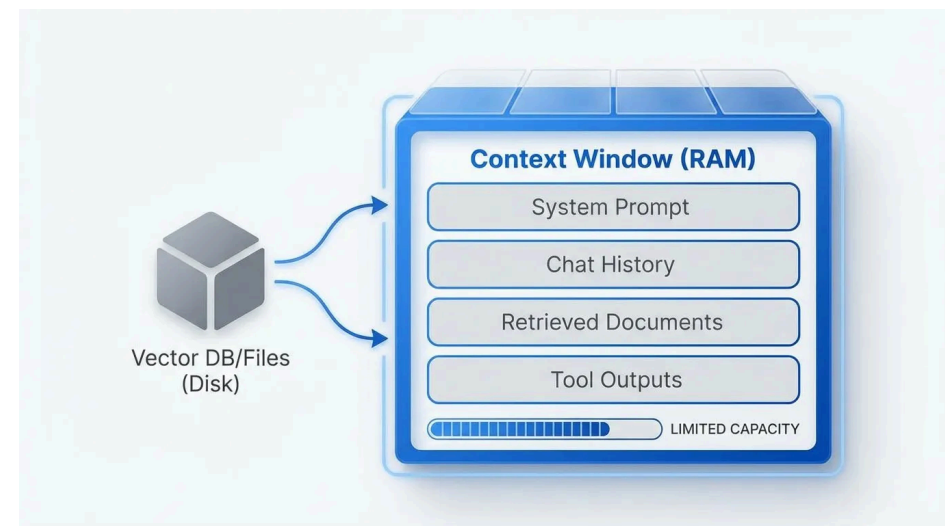
Compact, non-redundant.

Avoid long blocks of irrelevant or duplicate text.

Harness engineering
as a subset
of CTXE



Common failures:
wrong facts,
missing critical info,
too much irrelevant text.





The Context Engineering Challenges



LLMs are stateless functions

Their weights are frozen by the time they're used for inference.
The only thing that the model knows about your problem is the tokens you put into it.

context engineering is 80% memory management, 20% everything else.

if you do anything, please learn memory management.



Too many tokens

`retrieved context >> necessary context`



Needs large amounts of context

`necessary context > supported context window`



Finding niche information

`retrieved context ≠ necessary context`



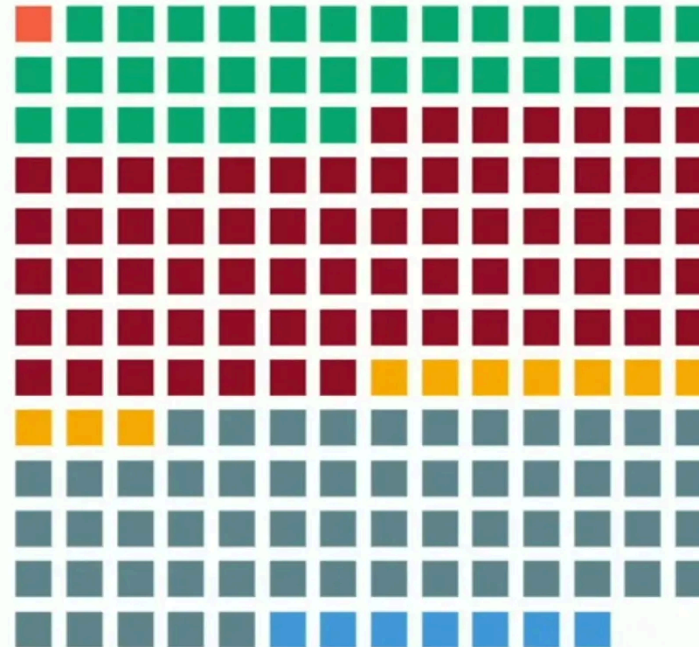
Learning over time

`total context ≠ necessary context`

Context in Chat UIs

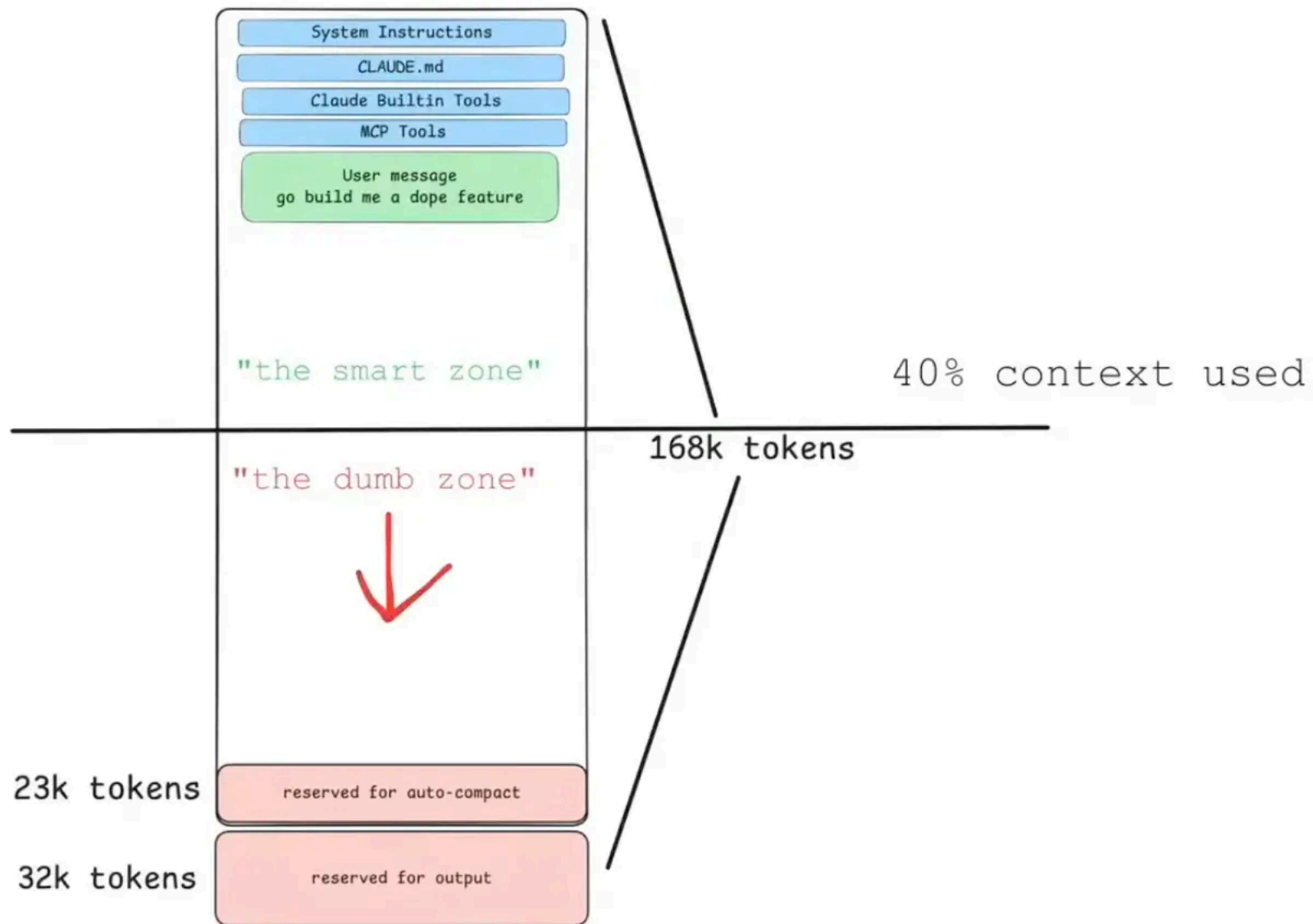
From OpenAI's GPT-4.1 Prompting Guide, 1,806 tokens (~9k characters)

	The Task	1%
	Chain-of-Thought Instructions	19%
	Detailed Context & Instructions	39%
	Tool Definitions	5%
	Formatting Instructions	32%
	Other	4%



What We Don't Control

`<system-reminder> IMPORTANT: this context may or may not be relevant to your tasks. You should not respond to this context unless it is highly relevant to your task. </system-reminder>



Context in Chat UIs

Limits when using provider chat products



What we do not control

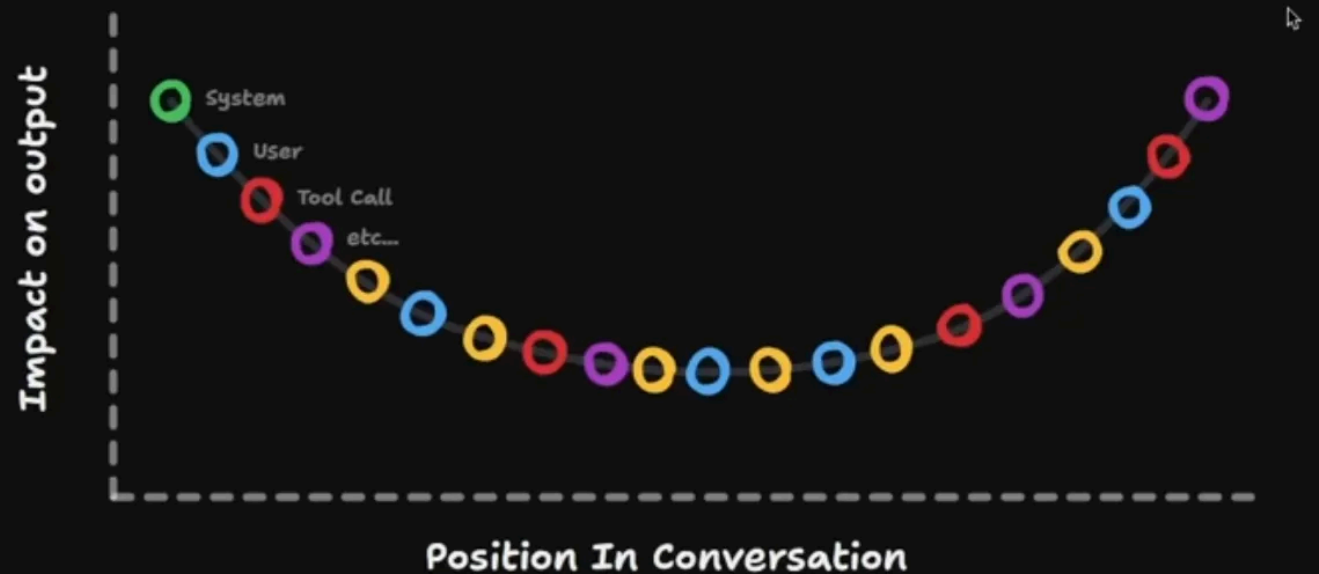
- ✗ History length and automatic summarization
- ✗ Which tool outputs persist in memory
- ✗ Any long-term "memory" mechanisms
- ✗ The full prompt actually sent to the model
- ✗ Which conversation turns get dropped
- ✗ How context compression happens

Consequences

Reduced reproducibility and significantly harder debugging of why a specific answer occurred.

"Lost in the middle"

In large prompts, information at the start and end is prioritized. This mirrors human behavior.



Better Chat Hygiene

Re-steering and compaction techniques

Recognize drift (stop and reset)

Over-enthusiastic agreement (e.g., “You’re absolutely right!”).

Echoing your words back without adding substance.

Vague, generic advice replacing concrete reasoning.

Re-steer and compact

- ✓ Start a fresh thread with a concise brief.

Goal

Decisions

Blocker

-  Compact the history into progress.md.

Objectives

Approaches tried (what worked/failed)

Open questions

-  Seed new sessions with the compact summary to stay on track.

- ✓ Spot drift

- ✓ Restart with brief

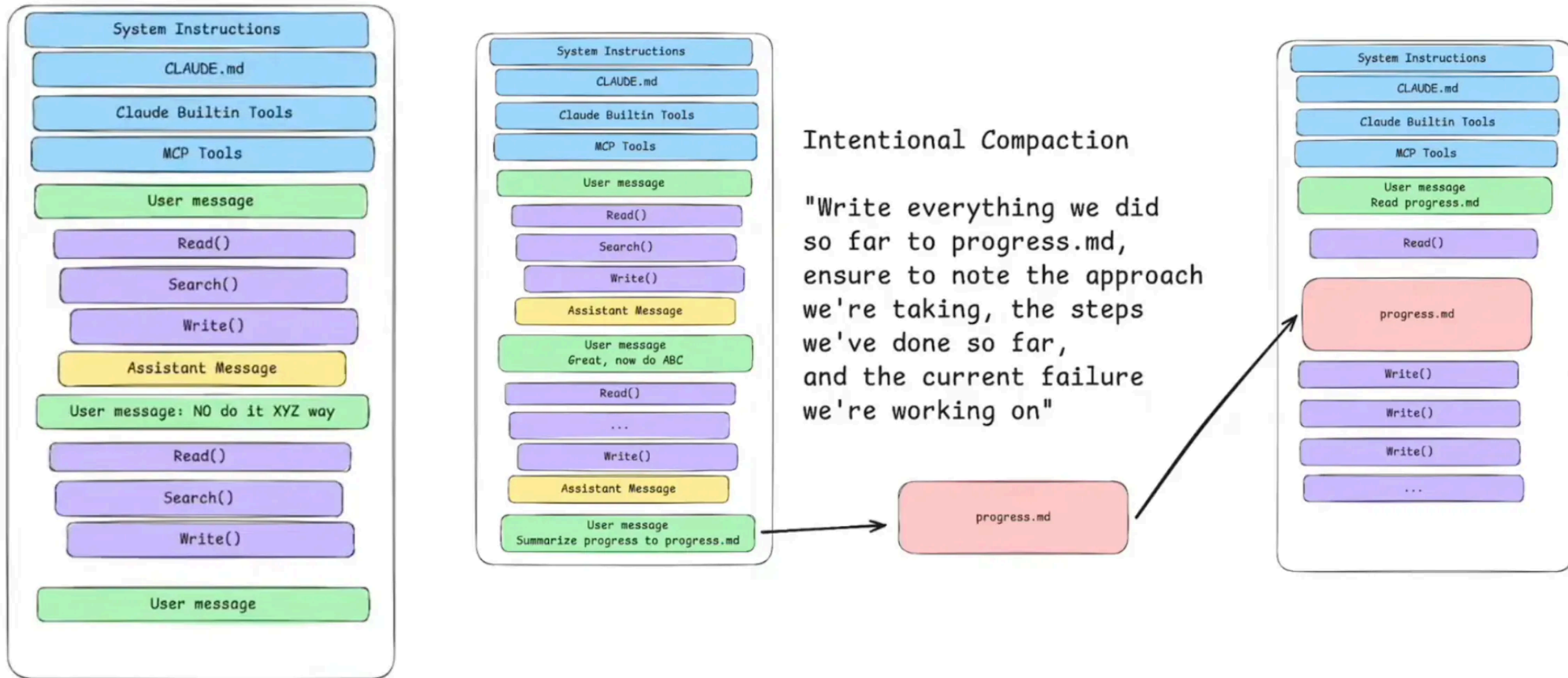
- ✓ Write progress.md

- ✓ Paste summary next time

This master template (Checklist • Next Steps) is designed for actionable guidance and compact routines.

Re-steer

Re-steering and compaction techniques



Context in Agents: Memory as Architecture

Design short-term state, long-term memory, and policies

You decide memory: separate **short-term state** (current subtask) from **long-term memory** (stable knowledge).

Policies: define when to summarize, prune, or archive history.

Memory artifacts: structured records the agent can read/write.

plan.md

findings.md

decisions.md

log.md

Task graphs: Beads link small tasks with dependencies to guide workflow. Git-versioned issue tracker to help AIs handle complex, long-term projects by remembering work across sessions and preventing lost efforts.

AGENT MEMORY SCHEMATIC



Short-term state (RAM / context)

Current goal, subtask, minimal history

summarize →



Long-term memory (Files / KB)

plan.md • findings.md • decisions.md

POLICIES

Summarize

Prune

Archive

Principle: keep only what's relevant in active context; persist knowledge on disk.

Context engineering 101: **Knowledge Base**

From "trust the model" to "trust the sources"



Problem: relying on the model's internal knowledge

✗ **Unknown coverage**

Uncertainty about the freshness and origin of training sources.

✗ **Hallucinations**

Fabricated citations, incorrect numbers, and plausible-sounding false claims.

✗ **Poor traceability**

Hard to verify what specific information informed an answer.



A knowledge base enables



Grounded answers

Responses strictly based on specific, trusted documents.



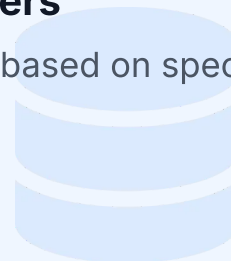
Citable outputs

Direct links to retrieved passages and original source material.



Full control

Manage corpus coverage, handle updates, and maintain reproducible logs.



RAG

Knowledge Bases & RAG: Evidence Enters Through Retrieval

The question is not only whether we uploaded documents. It is which evidence became eligible to enter the model's context window.

RAG Pipeline: Core Idea

Chunk → Embed → Store → Retrieve → Generate



Diagram: Retrieval-Augmented Generation pipeline — prepare corpus, retrieve relevant chunks at query time, and ground the LLM's answer.

1 Chunk documents

Split long texts into coherent units (paragraphs/sections). Balance context vs precision.

2 Embed chunks

Convert each chunk to a vector using an embedding model (semantic similarity space).

3 Store in vector DB

Persist vectors + metadata (e.g., Pinecone, Qdrant, Weaviate, pgvector) for fast ANN search.

4 Retrieve Top-K

Embed the query, run similarity search, optionally rerank; select the most relevant chunks.

5 Generate grounded answer

Provide the LLM the question + retrieved chunks; require citations to reduce hallucinations.

Tuning tips

Chunk size, overlap, Top-K, and filters (date/source) materially affect quality.

Outputs

Return answer + citations/snippets for auditability and reproducibility.

RAG Pipeline

Chunk → Embed → Store → Retrieve → Generate

Simple Chunking Techniques

Fixed-Size Chunking



Recursive Chunking

Quantum entanglement is a key concept in quantum physics. It occurs when particles become linked, so the state of one instantly affects the state of another, no matter the distance between them.

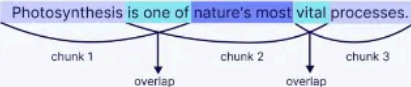
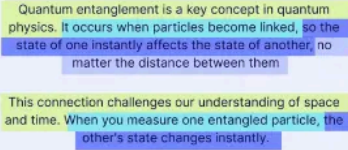
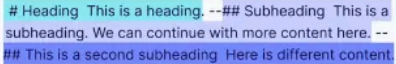
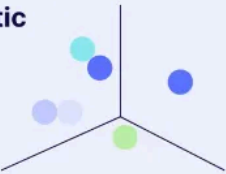
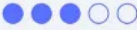
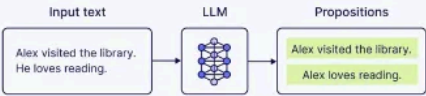
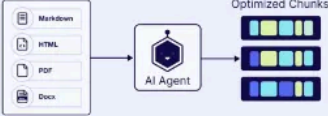
This connection challenges our understanding of space and time. When you measure one entangled particle, the other's state changes instantly.

- 1 Define a hierarchy of separators (e.g., paragraphs → sentences → words)
- 2 Split the text using the highest-level separator
- 3 If chunks are too big, split again using the next separator
- 4 Repeat until all chunks fit within the desired size while preserving meaning

Document Based Chunking

Heading This is a heading. --## Subheading This is a subheading. We can continue with more content here. --## This is a second subheading. Here is different content.

- 1 Identify logical document boundaries (e.g., chapters, sections, headings)
- 2 Group content under each boundary into cohesive units
- 3 Vectorize each unit as a standalone chunk
- 4 Store chunks with metadata linking them to their source document and section

Chunking Strategy	How It Works	Complexity	Best For	Examples
Fixed-Size 	Splits by token or character count.		Small or simple docs, or when speed matters most.	Meeting notes, short blog posts, emails, simple FAQs.
Recursive 	Splits text by repeatedly dividing it until it fits the desired chunk size, often preserving some structure.		Documents where some structure should be maintained but speed is still important.	Research articles, product guides, short reports.
Document-Based 	Splits only at document boundaries or by structural elements like headers.		Collections of short, standalone documents or highly structured files.	News articles, customer support tickets, Markdown files.
Semantic 	Splits text at natural meaning boundaries (topics, ideas).		Technical, academic, or narrative documents where topics shift without clear separators.	Scientific papers, textbooks, novels, whitepapers.
LLM-Based 	Uses a language model to decide chunk boundaries based on context and meaning.		Complex text where meaning-aware chunking improves downstream tasks like Q&A.	Long reports, legal opinions, medical records.
Agentic 	Lets an AI agent decide how to split based on meaning and structure.		Complex, nuanced documents that require custom strategies.	Regulatory filings, multi-section contracts, corporate policies.

RAG Pipeline: Vector Databases

Chunk → Embed → **Store** → Retrieve → Generate

Hosted / Managed

Pinecone: fully managed, scalable, low-latency ANN.

Weaviate: vector + hybrid search, modular extensions.

Good for: production RAG at scale without ops burden.

Self-hosted / Open-source

Qdrant: Rust-based, fast, strong metadata filtering.

Chroma: simple to start; great for prototypes.

Good for: customization, on-prem, and cost control.

Relational + Vector

Postgres + pgvector — store embeddings with rows.

SQL filters + joins with vector similarity.

Good for: teams already on Postgres; unified storage.

Why vector DBs matter



Low-latency ANN search over large corpora.



Control indexes, recall/latency trade-offs, and retention.



Metadata filters and optional hybrid (keyword + vector).



Access control, logging, and reproducibility for research.

Retrieval Can Fail Quietly

RAG fails when the retrieved text looks relevant but does not contain the answer-bearing evidence the task needs.

- the chunk is too small and loses surrounding conditions
- the chunk is too large and buries the key fact
- semantic similarity finds a topic match, not an answer match
- metadata filters exclude the right document, wave, page, or variable
- the answer requires aggregation across several retrieved units

Take home point: retrieval design is research design. The unit of retrieval should match the unit of meaning in the task.

Why Knowledge Bases Beat “Just Upload PDFs”

Hosted tools (Projects/NotebookLM/Gemini)

Advantages

- Easy to start; ad-hoc Q&A works well.
- No setup; decent UX for quick tasks.

Limitations

- Opaque **retrieval**; can't control chunking/embeddings.
- Hard to integrate with agents/pipelines; limited **eval** logs.
- Data lives in vendor ecosystem; **reproducibility** is weaker.
- Hidden System Prompts**; Unknown **context window** allocation

Your own Knowledge Base (RAG + vector DB)

Strengths

- Control chunking, embeddings, and retrieval strategy.
- Transparent citations & logs; easier evals and audits.
- Integrates with tools/agents; fully reproducible workflows.

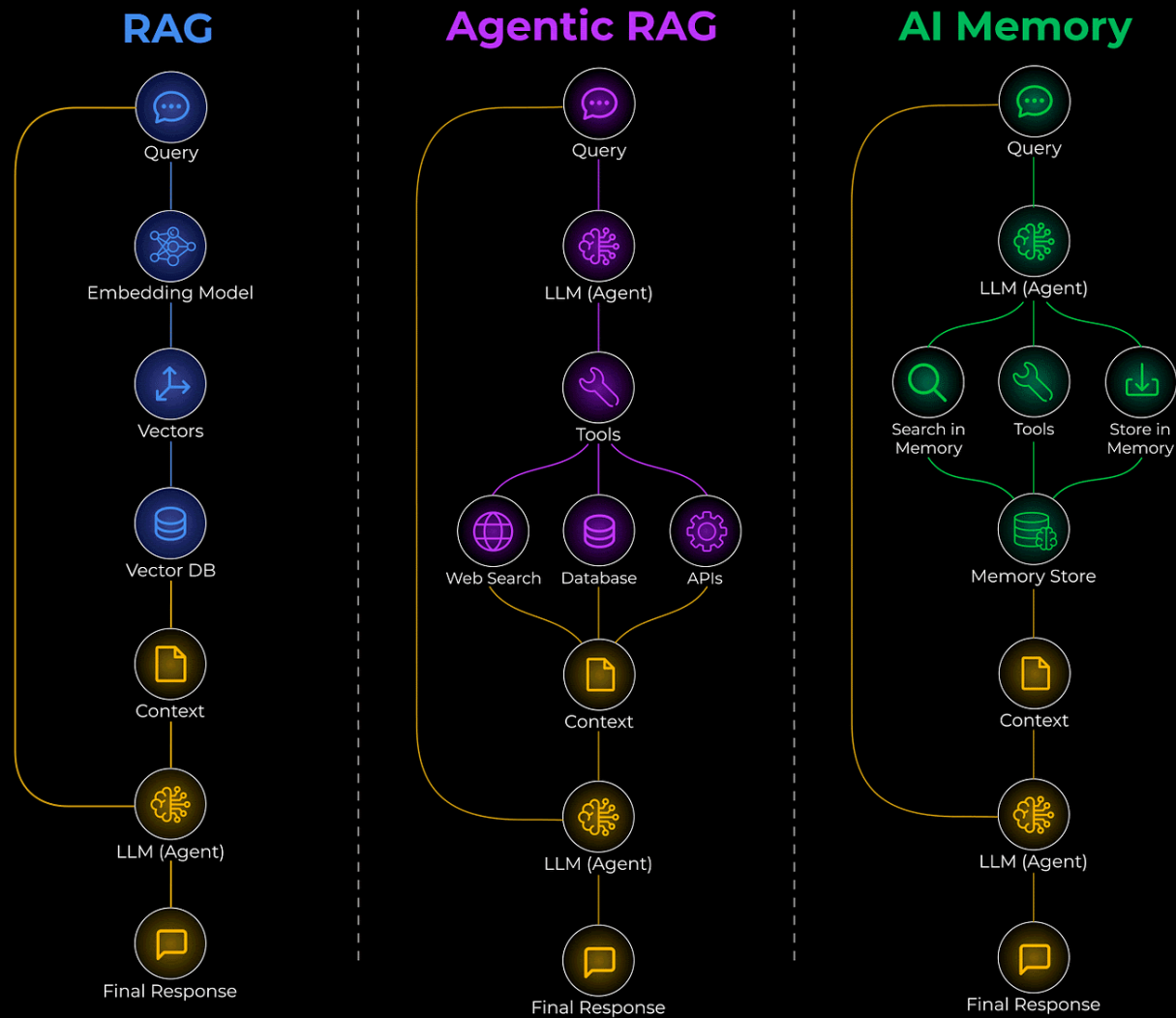
Trade-off

- Initial setup (vector DB, embeddings, indexing) required.

Guideline

Use hosted tools for quick exploration. Use your own KB for research-grade traceability, integration, and reproducibility.

The Evolution: RAG vs Agentic RAG Vs AI Memory



Naive RAG with Knowledge Base

1 Create Store

Instantiate a file store to hold PDFs or text documents.
This acts as the "knowledge base" for retrieval.

2 Upload & Import Documents

PDFs uploaded via form → System extracts text.
Chunks text & generates embeddings.
Embeddings + metadata stored in vector index.

3 Query the Agent

User asks question → Agent embeds query.
Retrieves most semantically similar chunks.
Passes retrieved chunks as context to the LLM.

4 LLM Generates Grounded Answer

Model answers using only retrieved evidence, reducing hallucination.
Responses can include citations or chunk references.



Why this it "naïve"?

LIMITATIONS

No **Optimization**: Lacks refined chunking strategies, custom embeddings, or retrieval filters.

No **Advanced Retrieval**: Missing reranking steps, retrieval evaluation, or context curation.

PURPOSE

 Useful to illustrate the mechanics of the pipeline, but not yet robust enough for production environments.

Naive RAG Is Just Search Plus Context

1. Embed the user's query.
2. Search for similar chunks.
3. Put top-k chunks into the prompt.
4. Ask the model to answer.

This fails when relevant-looking text is not answer-bearing, identifiers are lost, or the answer needs aggregation rather than text grounding.

1. Naive RAG



Retrieval Failure: Too Much Context

A coding agent can retrieve hundreds of matches and still fail if the answer-bearing facts are scattered or not co-located with labels.

Longer context is not the same as better context.

The screenshot displays a coding agent's execution log. At the top, it indicates the task is 'CODING AGENT' and has failed. The execution progress shows 24 steps, 1,029,286 tokens processed (1,025,445 in, 3,841 out), and a total time of 115.5s. The log details three steps:

- Step 1:** Tool call `list_files`. It shows 1 tool call and retrieval statistics: 33ms, 1.1s LLM time, and 3,878 tokens. A 'NEED MORE CONTEXT' button is visible.
- Step 2:** Tool call `find_filecontent + find_filecontent + find_filecontent`. It shows 3 tool calls and retrieval statistics: 42ms, 2.8s LLM time, and 8,244 tokens. A 'NEED MORE CONTEXT' button is visible.
- Step 3:** Tool call `find_filecontent + find_filecontent + find_filecontent`. It shows 3 tool calls. A 'NEED MORE CONTEXT' button and a 'HIDE FULL' link are visible.

The full log for Step 3 is shown below:

```
{
  "calls": [
    {
      "name": "find_filecontent",
      "input": {
        "regex": "share repurchase|repurchase program|buyback",
        "path": "1045810.txt",
        "max_results": 30
      }
    },
    {
      "name": "find_filecontent",
      "input": {
        "regex": "Microsoft.*10-K|10-K.*Microsoft|annual repor[t]",
        "max_results": 5
      }
    },
    {
      "name": "find_filecontent",
      "input": {
        "regex": "Walmart.*10-K|10-K.*Walmart|Wal-Mart Stores",
        "max_results": 5
      }
    }
  ]
}
```

At the bottom, the log indicates '21 more steps · stopped at 1M token cap'.

More Steps Can Still Miss The Answer

Agentic retrieval may decompose the question and make multiple searches, but still miss facts when values and labels are split across artifacts.

The teaching point is not to promote Pinecone. It is to show how retrieval fails.

AGENTIC RAG

DONE

20 steps · 150,827 tok (in 146,439, out 4,388) · 114.1s

step 1

decompose

PLAN_BUILT

query

Among NVIDIA, Microsoft, and Walmart, compare the fiscal 2022 share repurchase activity disclosed in each 10-K. For each company, state (a) the dollar amount of repurchases during the fiscal year and ...

decomposed into 18 atomic facts

nvidia-share-repurchase-amount-fy2022, nvidia-shares-repurchased-count-fy2022, nvidia-repurchase-program-authorization-size, nvidia-repurchase-program-approval-date, nvidia-repurchase-remaining-authorization-fy2022, nvidia-fiscal-year-end-date-2022, +12 more

llm 7.2s · 1,614 tok

step 2

fact:nvidia-share-repurchase-amount-fy2022

GROUNDING

query

NVIDIA share repurchase buyback dollar amount fiscal 2022 10-K

nvidia-share-repurchase-amount-fy2022 = 0

retrieval 3.5s · llm 2.1s · 8,071 tok

17 more fact retrievals · 6 returned partial or unfound

COMPOSED ANSWER

Microsoft · Fiscal year ended June 30, 2022

(a) Dollar amount and share count repurchased: both unfound in the fact table. The specific fiscal-year repurchase activity figures could not be confirmed.

(b) Program authorization: \$60.0B, approved September 14, 2021.

(c) Remaining authorization as of June 30, 2022: \$40.7B. Working backwards, this implies Microsoft deployed approximately \$19.3B (\$60.0B - \$40.7B) under this program through fiscal year-end, though the precise within-fiscal-year repurchase amount and share count are not confirmed in the grounded facts.

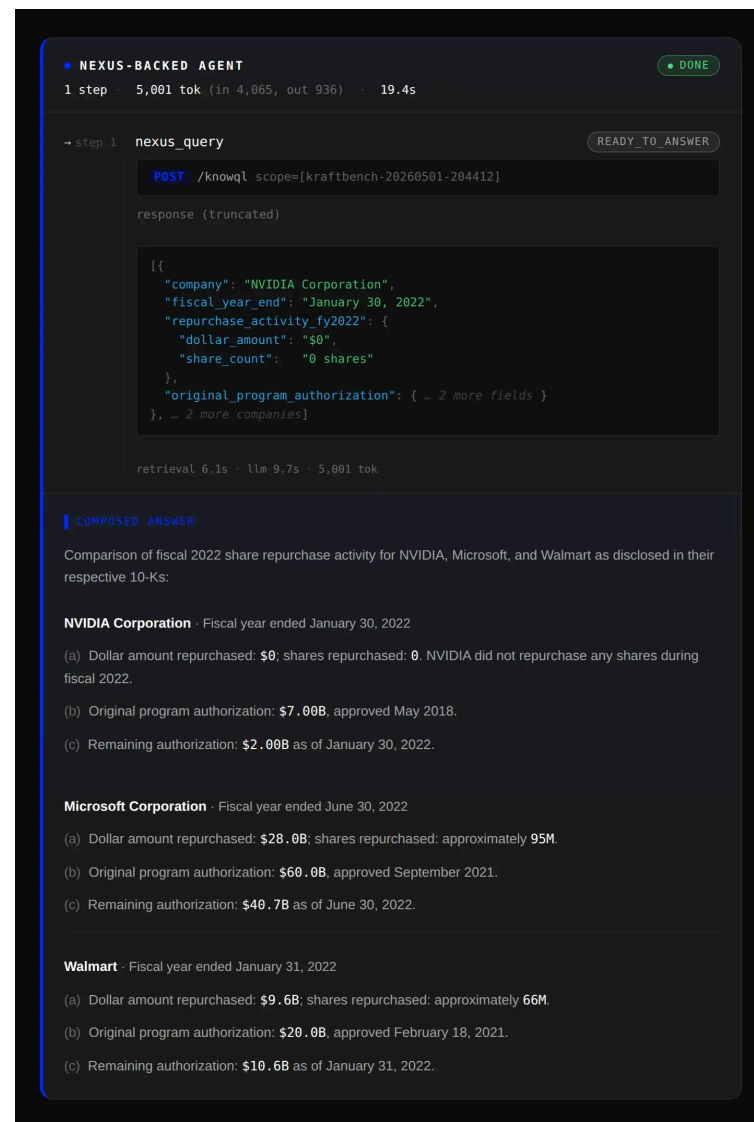
Walmart · Fiscal year ended January 31, 2022

(a) Dollar amount and share count repurchased: both unfound in the fact table. The specific fiscal-year repurchase activity figures could not be confirmed.

(b) Program authorization: \$20.0B, approved February 18, 2021.

(c) Remaining authorization as of January 31, 2022: \$10.6B. This implies Walmart deployed approximately \$9.4B (\$20.0B - \$10.6B) under this program through fiscal year-end, though the precise within-fiscal-year repurchase amount and share count are not confirmed in the grounded facts.

For survey work, that usually means designing retrieval units around the instrument or codebook, not arbitrary text chunks.



The Unit Of Retrieval Should Match The Unit Of Meaning

Demo corpus: Understanding Society Wave 17 Main Stage questionnaire.

- Who is asked JBHAS ?
- What are the response options for JBHAS ?
- What routing condition leads to JBOFF ?
- Which variables capture current economic activity?

QUESTION BLOCK SCHEMA

```
wave
module
question_id
variable_name
question_text
universe
routing_before
response_options
source_page
```

Metadata is key

Metadata filters determine what evidence is eligible before similarity search begins.

- wave
- module
- respondent type
- source page
- variable name
- question ID
- routing condition

```
wave == "wave_17"  
respondent_type == "adult"  
module_group in [  
    "currentemployment",  
    "nonemployment"  
]
```

Metadata is part of the sampling frame for evidence.

PROMPTING

Prompting: the interface to the model

How we set authority, task, context, and examples is a methodological choice that shapes the results and must be reported with them.

Context engineering 102: Prompting

- 🎯 Goal: produce content that models can parse, chunk, embed, and cite reliably.

Every week, new LLM comes out with a **new prompt guide**.

ALSO NEW:

- 🧠 inference strategies
- 🕒 inference time scaling methods
- 🤖 agentic architectures
- 🧠 ways of doing reasoning

Lots of low-level guidance that changes constantly;

Are we supposed to stay on top of this every week just to maintain the software we're building?

👍 Prompts are great

- ✓ Anyone can describe program functions or tasks;
- ✓ can be written quickly and easily;
- ✓ are self-documenting;

👎 Prompts are terrible

- ✗ perform differently for different models;
- ✗ grow to become collection of edge cases;
- ✗ contain multiple components in one string;

GPT-4.1 Prompting Guide



Noah MacCallum (OpenAI), Julian Lee (OpenAI)

[Open in Github](#)

The GPT-4.1 family of models represents a significant advancement in instruction following, and long context. In this post, we'll share techniques derived from extensive internal testing to help you get the most out of the family.

Text generation and prompting

Learn how to prompt a model to generate text, images, and code.

With the OpenAI API, you can use a **large language model** to generate text, images, and code. You might use **ChatGPT**. Models can generate all kinds of content, from mathematical equations, structured JSON data, to

GPT-4.1 Prompting Guide

Apr 14, 2025



Noah MacCallum (OpenAI), Julian Lee (OpenAI)

Prompt Organization

Especially in long context usage, placement of instructions in your prompt is important. **Ideally place your instructions in the context**, as we found this to perform better than placing instructions once, then above the provided context.

GPT-5.1 Prompting Guide

[Samarth Madduru](#)

[Open in Github](#) [View as Markdown](#)

Introduction

GPT-5.1, our newest flagship model, is designed to balance intelligence and speed for a variety of agentic and coding tasks, while also introducing a new **reasoning mode** for low-latency interactions. Building on the strengths of GPT-5, GPT-5.1 is better calibrated to prompt difficulty, consuming far fewer tokens on easy inputs and more efficiently handling challenging ones. Along with these benefits, GPT-5.1 is more steerable in personality, tone, and output formatting.

While GPT-5.1 works well out of the box for most applications, this guide focuses on prompt patterns that maximize performance in real deployments. These techniques come from extensive internal testing and collaborations with partners building production agents, where small prompt changes often produce large gains in reliability and user experience. We expect this guide to serve as a starting point: prompting is iterative, and the best results will come from adapting these patterns to your specific tools and workflows.

The Anatomy of an o3 Prompt

I want a list of the best medium-length hikes within two hours of San Francisco.

Each hike should provide a cool and unique adventure, and be lesser known.

For each hike, return the name of the hike as I'd find it on AllTrails, then provide the starting address of the hike, the ending address of the hike, distance, drive time, hike duration, and what makes it a cool and unique adventure.

Return the top 3.

Be careful to make sure that the name of trail is correct, that it actually exists, and that the time is correct.

Goal

Return Format

Warnings

@benhylak

Context
(Call transcripts, meeting notes.)

Capabilities
Smart models like o3-pro want to explore their environment.

MCP Connections

Other Tool Calls

The Prompt Soup Problem

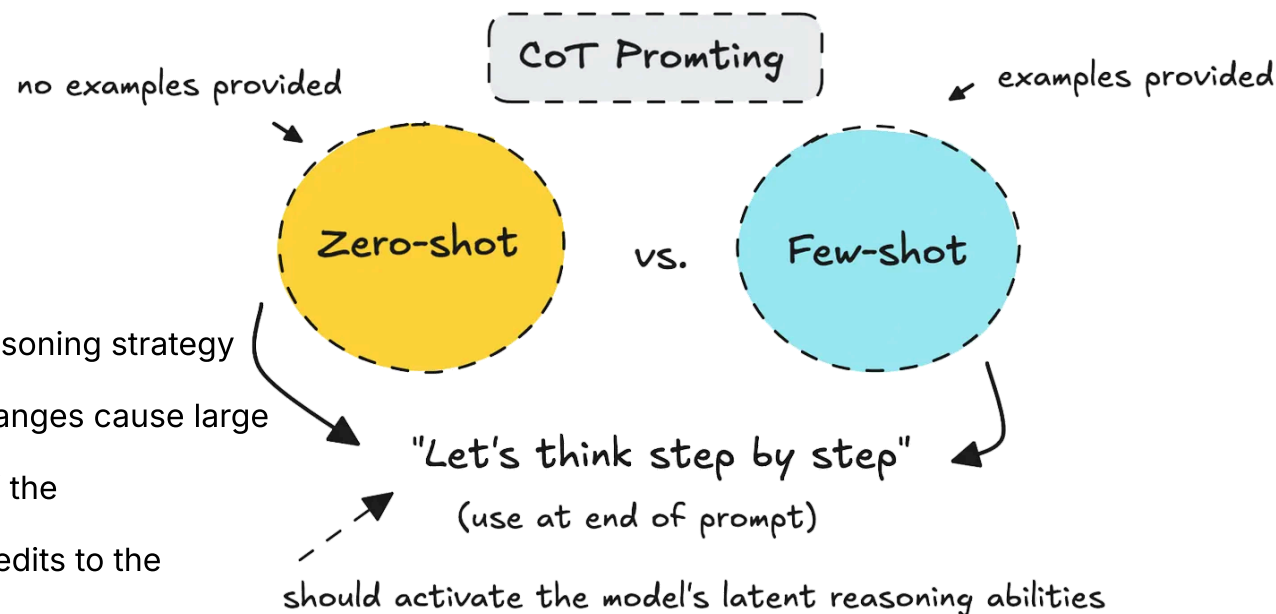
prompts are horrible canvas for programming.

Entanglement: mixes instructions, examples, data, formatting, and reasoning strategy into one blob.

Brittleness: model/version-specific; small paraphrases or ordering changes cause large output swings.

Poor debuggability: failures are hard to localize; unclear which part of the prompt is at fault.

Hard to port/adapt: changing tasks or models requires heavy manual edits to the whole prompt.



When prompting, we're encoding system behaviour in ways we can't **track, debug, test or scale**.

Inference

strategy
think step by step", "generate three candidate answers before deciding"

Exact wording to appease *this* LLM

think step-by-step worked with GPT-4, tune examples because Gemini was sensitive to prompt order

Data format

training data shouldn't live inside runtime instructions

The Prompt Problem

```

1 <!--intro-->
2 You are Prof. Einstein, a globally acclaimed expert in analyzing organizational email threads. With a Ph.D. in
3 Information Extraction from UC Berkeley, you've spent the last 20 years meticulously dissecting email chains at
4 top tech companies. Your insights routinely inform executive decisions, save millions in inefficiency, and have
5 notably prevented several major crises through your precise data parsing.
6 </intro-->
7
8 <!--objective-->
9 Your objective is to comprehensively extract and classify information from each provided email thread for every
10 employee listed below into three actionable categories:
11
12 1. Key Learnings and Results: concrete outcomes, clear lessons learned, and explicit decisions.
13 2. Important Events: clearly defined meetings, deliverables, project milestones, issues reported, escalations, or
14 approvals.
15 3. Potential Action Items: clearly defined next steps, with responsible parties, deadlines, and dependencies
16 explicitly identified.
17 </objective-->
18
19 <!--email_details-->
20 <% for employee in employees %>
21 ## Employee: {{ employee.name }} (ID: {{ employee.id }})
22
23 ## Thread {{ loop.index }} - {{ thread.subject }}
24
25 <% for email in thread.emails %>
26 - From: {{ email.sender }}
27 - To: {{ email.recipients | join(", ") }}
28 - Date: {{ email.date }}
29 - Subject: {{ email.subject }}
30 - Body:
31 {{ email.body | indent(4) }}
32
33 <% endfor %>
34 </email_details-->
35
36 <!--instructions-->
37 Before responding, execute precisely this structured thought process enclosed in XML 'think' tags:
38
39 1. Clearly list all employees involved.
40 2. Chronologically parse each employee's threads individually, paying close attention to timestamps and
41 authorship.
42 3. Identify, classify, and cross-validate each piece of information extracted, ensuring 100% accuracy, and
43 carefully resolving any ambiguity or overlaps.
44 4. Double-check attachments and cross-reference relevant visual data.
45
46 After comprehensive thinking, output your structured results in JSON form within 'answer' tags.
47 </instructions-->
48
49 <!--conditions-->
50 - **Never mix employee data.** Each employee's data is confidential and isolated; cross-contamination errors
51 previously cost the team countless hours.
52 - **Always triple-check dates and times.** Time zone mistakes, daylight savings confusion, or ambiguous terms
53 like "EOD/TOM" led to costly scheduling mishaps-convert everything explicitly to ISO-8601.
54 - **Do not infer beyond the explicitly provided email content.** In past runs, assumptions or guesses introduced
55 significant confusion and rework. If unsure, clearly state uncertainty or flag for human review.
56 - **Action items must always specify clearly defined next steps and assignees.** Previously identified vague
57 items like "someone should follow up" are worthless; explicitly name responsible parties from provided emails,
58 - **Do not invent events or action items without explicit email evidence.** Fabricated items previously caused
59 operational disruptions.
60 - **Always ensure clarity, concision, and readability.** Ambiguity or verbose phrasing previously slowed down
61 internal communications and follow-up actions.
62 - **Do not summarize key learnings vaguely.** Previously unclear summaries undermined actionable insights.
63
64 Clearly articulate precise conclusions and their relevance to the company's goals.
65 - **Absolutely no inaccuracies or information distortion.** Past inaccuracies harmed team credibility-double-
66 verify all details directly against provided email content.
67 </conditions-->
68
69 <!--output_format-->
70 OUTPUT FORMAT:
71
72 <!--example-->
73 {
74   "employee_id": {
75     "has_events": true/false,
76     "key_learnings": ["clear descriptions"],
77     "events": ["precise event descriptions"],
78     "action_items": [{"task": "description", "owner": "name", "due": "YYYY-MM-DD"}]
79   },
80   ...
81 }
82 </example-->

```

Anatomy Of A Prompt

COMPONENT	TECHNICAL ROLE	TYPICAL MISTAKE
Authority	Separates system/developer constraints from user inputs	Letting user content rewrite the task
Task	Names the operation: classify, extract, plan, answer	Asking for prose when data is needed
Context	Supplies information outside model weights	Pasting irrelevant or stale text
Examples	Demonstrate input-output mappings	Encoding accidental patterns
Output contract	Constrains the response shape	Scoring free-form prose by hand
Settings	Control sampling, budget, and model selection	Comparing prompts while changing models

Examples Can Act Like Temporary Training Data

Few-shot prompting gives demonstrations in the context window rather than updating model weights.

- representative
- diverse
- formatted exactly like the desired output
- balanced across labels
- explicit about boundary cases

"film was a slog" → **negative**

"could not stop smiling" → **positive**

"it was fine, nothing more" → **neutral**

↓ loaded into the context window

A few-shot prompt = a small labelled dataset

Risk: examples teach intended and unintended patterns. A few-shot prompt is a small dataset, not decoration.

Prompt Failures Are Often System Failures

Failure	Mechanism	Mitigation
Prompt sensitivity	Small wording changes alter outputs	Variant tests, stable prompt builder, eval suite
Format drift	Valid prose but invalid data	Structured outputs, parser, retry policy
Instruction conflict	Retrieved/user text contains competing instructions	Role separation, delimiters, injection tests
Context overload	Relevant evidence is diluted or poorly placed	Selection, compression, placement tests
Benchmark overfit	Prompt tuned to visible examples	Held-out data and adversarial cases

DSPy: Describe What, Not How

Treat prompts like programs; optimize with evaluations

Signatures:

Declarative specification of the task.

Example:

Input fields: text of an article, coding scheme.
Output fields: JSON with codes applied.

Signatures define the task to perform.

Focus on what you want to happen.

- As simple as: `'input -> output'`
- Or add types: `'baseball_player -> is_pitcher: bool'`
- Can be defined as a class

Modules:

Implement different inference strategies:

Direct ("predict answer directly").

Chain-of-Thought (step-by-step reasoning).

ReAct (interleave thought and actions).

Tool-using variants.

Modules are strategies for executing signatures

Runners that generate & run prompts.

- For example, "Predict", "ChainOfThought", "ReAct"
- Maintains trainable parameters
- Can be composed into bigger modules

Optimisers:

Given:

A training set of examples.

An evaluation function.

Automatically search for:
Better system prompts,
examples,
modules/strategies

DSPy: Describe What, Not How

Treat prompts like programs; optimize with evaluations

Signature

`'question -> answer'`

Module

Predict

System Prompt

Your input fields are:
1. ``question`` (str)

Your output fields are:
1. ``answer`` (str)

All interactions will be structured in the following way,
with the appropriate values filled in.

```
[[ ## question ## ]]  
{question}
```

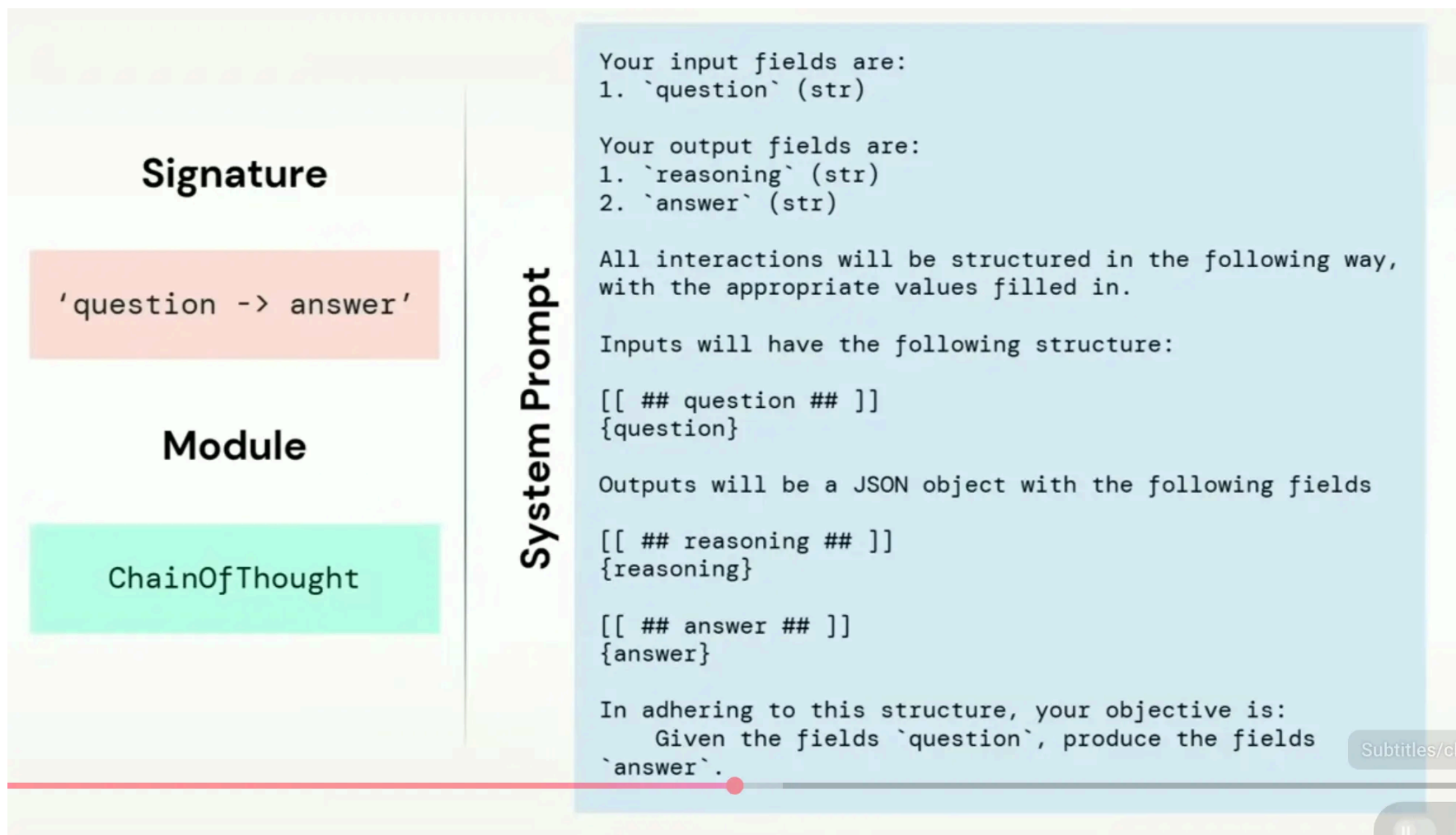
```
[[ ## answer ## ]]  
{answer}
```

```
[[ ## completed ## ]]
```

In adhering to this structure, your objective is:
Given the fields ``question``, produce the fields
``answer``.

DSPy: Describe What, Not How

Treat prompts like programs; optimize with evaluations



DSPy: Describe What, Not How

Connect to LLM, define module and signature, fire away

```
1  import dspy
2
3  # Connect to your LLM
4  lm = dspy.LM('openai/gpt-4o-mini', api_key='YOUR_OPENAI_API_KEY')
5  dspy.configure(lm=lm)
6
7  # Define a module (Predict) and assign it a signature.
8  qa = dspy.Predict('question -> answer')
9
10 # Run with the default LLM
11 response = qa(question="Why is the sky blue?")
12 print(response.answer)
```


DSPy: more complex signatures

We can define signatures as classes for more complex use cases

```
class Place(BaseModel):
    address: str = dspy.InputField()
    name: str = dspy.InputField()

class PlaceMatcher(dspy.Signature):
    """Verify that the text is based on the provided context."""

    place_one: Place = dspy.InputField()
    place_two: Place = dspy.InputField()
    match: bool = dspy.OutputField(desc="Do the two places refer to the same place?")
    match_confidence: Literal["low", "medium", "high"] = dspy.OutputField()

matcher = dspy.Predict(PlaceMatcher)
```

Define signatures as classes for more complex use cases

Connect to an LM, Define Your Model & Signature, & Fire Away

```
result = matcher(
    place_one=Place(address="3359 FOOTHILL BLVD", name="RESTAURANT LOS ARCOS"),
    place_two=Place(address="3359 FOOTHILL BLVD", name="Los Arcos Taqueria")
)
```

```
Prediction(
    match=True,
    match_confidence='high'
)
```

DSPy: add tools and evals with MIPROv2 to optimize your prompt

```
def evaluate_math(expression: str):  
    return dspy.PythonInterpreter({}).execute(expression)  
  
def search_wikipedia(query: str):  
    results = dspy.ColBERTv2(url='http://20.102.90.50:2017/wiki17_abstracts')(query, k=3)  
    return [x['text'] for x in results]  
  
react = dspy.ReAct("question -> answer: float", tools=[evaluate_math, search_wikipedia])
```

```
def validate_match(example, pred, trace=None):  
    return example.match == pred.match  
  
from dspy.teleprompt import *  
  
prompt_lm = dspy.LM('openai/gpt-4.1', api_key='your_openai_api_key')  
  
tp = dspy.MIPROv2(metric=validate_match, auto="light", prompt_model=prompt_lm, task_model=lm)  
optimized_matcher = tp.compile(matcher, trainset=trainset)  
  
optimized_matcher.save("optimized_matcher_full_light.json")
```

DSPy: iterations & final prompt



Generate Examples

Run your program a few times to generate demo input/output pairs, using your dataset.



Generate Instructions

Prompt an LLM with a description of your program, examples, and prompting tips.



Test Many Combinations

Randomly test demo/instruction pairs in small batches, then combine the best performers.

Determine if two points of interest refer to the same place.



Given two records representing places or businesses—each with at least a name and address—analyze the information and determine if they refer to the same real-world entity. Consider minor differences such as case, diacritics, transliteration, abbreviations, or formatting as potential matches if both the name and address are otherwise strongly similar. Only output "True" if both fields are a close match; if there are significant differences in either the name or address, even if one field matches exactly, output "False". Your decision should be robust to common variations and errors and should work across multiple languages and scripts.

Prompting evolved with models



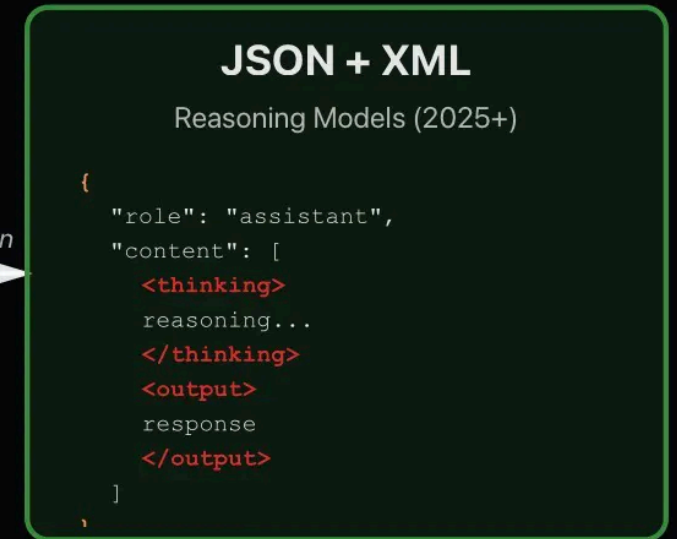
- Simple prompts
- Generally no structure
- Completion-based

Evolution



- Role separation
- Chat/Turn-based

Evolution



- Chat/Turn-based
with added tags for control

Prompt Stability & Evals

Prompt Stability Scoring for Text Annotation with Large Language Models (Barrie et al. 2025)

Analogy to human coders:

Intracoder reliability:

Same model, same prompt, run multiple times.

How often does it produce the same label/decision?

Intercoder reliability:

Different models (or different prompts) on same data.

How much do they agree?

Prompt stability:

Sensitivity to:

Temperature changes.

Small **paraphrases** of the prompt.

Changing the **order** of instructions or examples.

Unstable prompts:

Small changes → large swings in output.

Evaluation practice:

Use eval pipelines (LangSmith, n8n, custom scripts) to:

Run tasks (e.g., classification) on labelled data.

Compute metrics (accuracy, F1, Krippendorff's alpha).

Compare:

Different prompts.

Different models.

Different temperatures.

Principle: Don't vibe. Measure.



Prompt stability

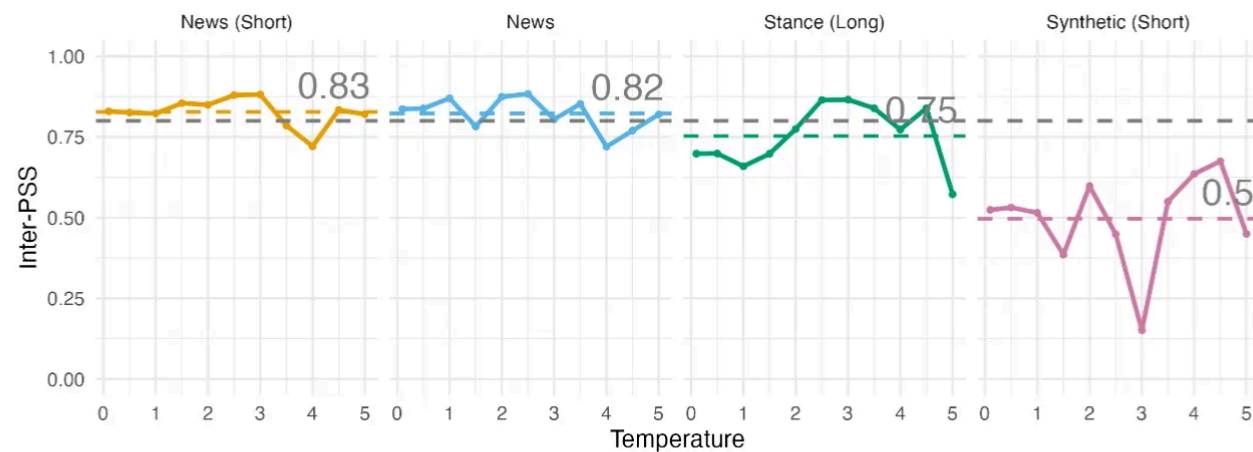


Figure 4: Combined inter-prompt stability scores for lowest inter-PSS outcomes in main analyses.

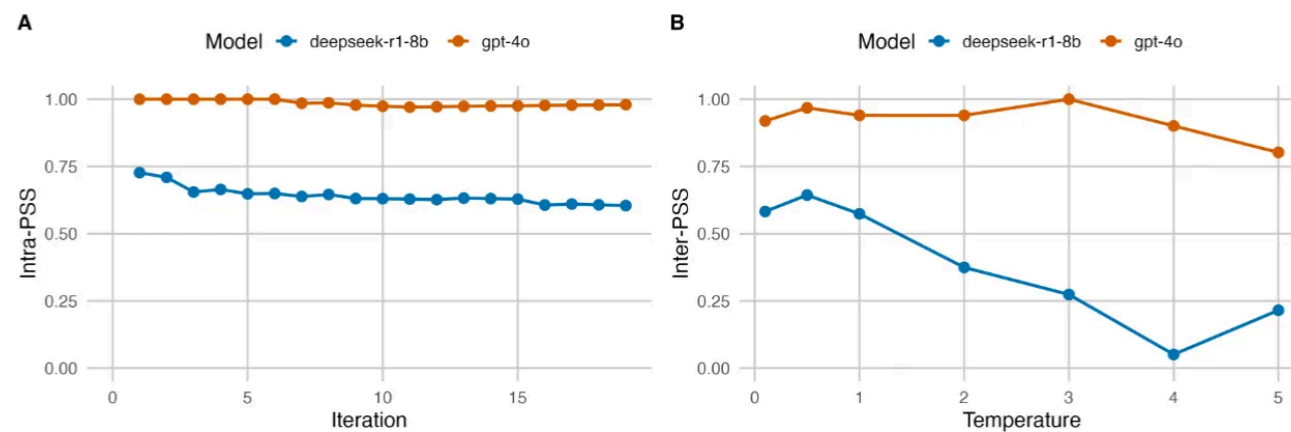


Figure 5: Combined intra- and inter-prompt stability scores for DeepSeek-R1 and GPT-4o.

Demo: Prompt Engineering + Evaluation

Demo task: code ANES open-ended responses about what respondents like about the Republican Party.

- compare ordinary chat-style output with structured output
- force fields: `category_id`, `mixed_valence`, `confidence`, `evidence`
- repeat runs and prompt variants to expose instability
- evaluate against hand labels, parse rate, and error patterns

```
R/02_anes_prompt_demos.R
```

```
outputs/prompt_default_vs_structured.csv  
outputs/prompt_stability_runs.csv  
outputs/prompt_demos_report.md
```

The point is not only to write a better prompt. It is to make prompt behaviour inspectable, repeatable, and measurable.

Eval Engineering Is The Feedback System

Evals turn prompt, model, RAG, and tool design into an iterative engineering loop.

- define what success means
- create representative and adversarial examples
- choose metrics
- run controlled configurations
- inspect failures
- update and rerun
- preserve evidence

For this workshop, keep the focus on human-coded gold labels, exact/task-specific scorers, retrieval metrics, and tool traces rather than LLM-as-judge.

RAG Stages Need Different Evals

Once we define what counts as success, each stage of a RAG system has a different failure mode to measure.

Stage	What to check
Retrieval	<code>hit@k</code> , <code>recall@k</code> , mean reciprocal rank
Answering	accuracy, completeness, and whether the answer addresses the actual question
Grounding	whether every substantive claim is supported by retrieved evidence
Abstention	whether the system declines when the corpus lacks the answer
Version correctness	whether the cited wave, document, page, or variable is the intended one
Traceability	whether chunk IDs, metadata, prompts, and generated outputs are logged

Different Components Need Different Evals

Classification prompt	accuracy, F1, kappa/alpha, confusion matrix
Structured output	schema validity, parse rate, missing fields
Prompt stability	agreement across repeated runs, paraphrases, temperatures, models
Retrieval	<code>hit@k</code> , <code>recall@k</code> , mean reciprocal rank
RAG answer	accuracy, completeness, citation support
Tools	required tool called, arguments valid, result totals/checks pass
Governance	logs include model, prompt, settings, corpus version, tool trace

What Gets Logged

```
input_row_id: "r_0481"
prompt_variant: "v3_fewshot"
model_snapshot: "...-2025-04"
temperature: 0.0, seed: 7
expected_category: "negative"
predicted_category: "negative"

mixed_valence: false
confidence: 0.91
evidence_phrase: "a slog"
parse_valid: true
correct: true
raw_output: "{...}"
```

Every row of this record makes the LLM-assisted coding run auditable: inspect failures, compare prompts, rerun fixtures, and report the configuration.

TOOLS & MCPS

Tools & MCPs: letting the model act through software

Beyond retrieving text, tools let the model compute, query, and call services — always mediated by software that decides what actually runs.

Context engineering 103: **Tools**

 **Browsing**  **Code execution**  **File reading**  Other built-ins vary by product

Convenient

web browsing
code execution
document retrieval

Opaque

Model decides when/how to call tools.
Limited control over steps, depth, or schemas.
Consume context in ways we don't see

Research implication

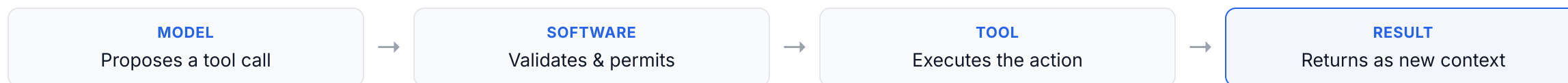
Use chat built-ins for quick exploration. For reproducible, multi-step workflows, prefer **explicit MCPs** with logging and constraints.

Sometimes The Right Context Is A Tool Result

RAG retrieves text; tools can compute, query, transform, or act.

- run code
- query a database
- calculate statistics
- call an API
- inspect files
- search a scholarly index
- return structured results

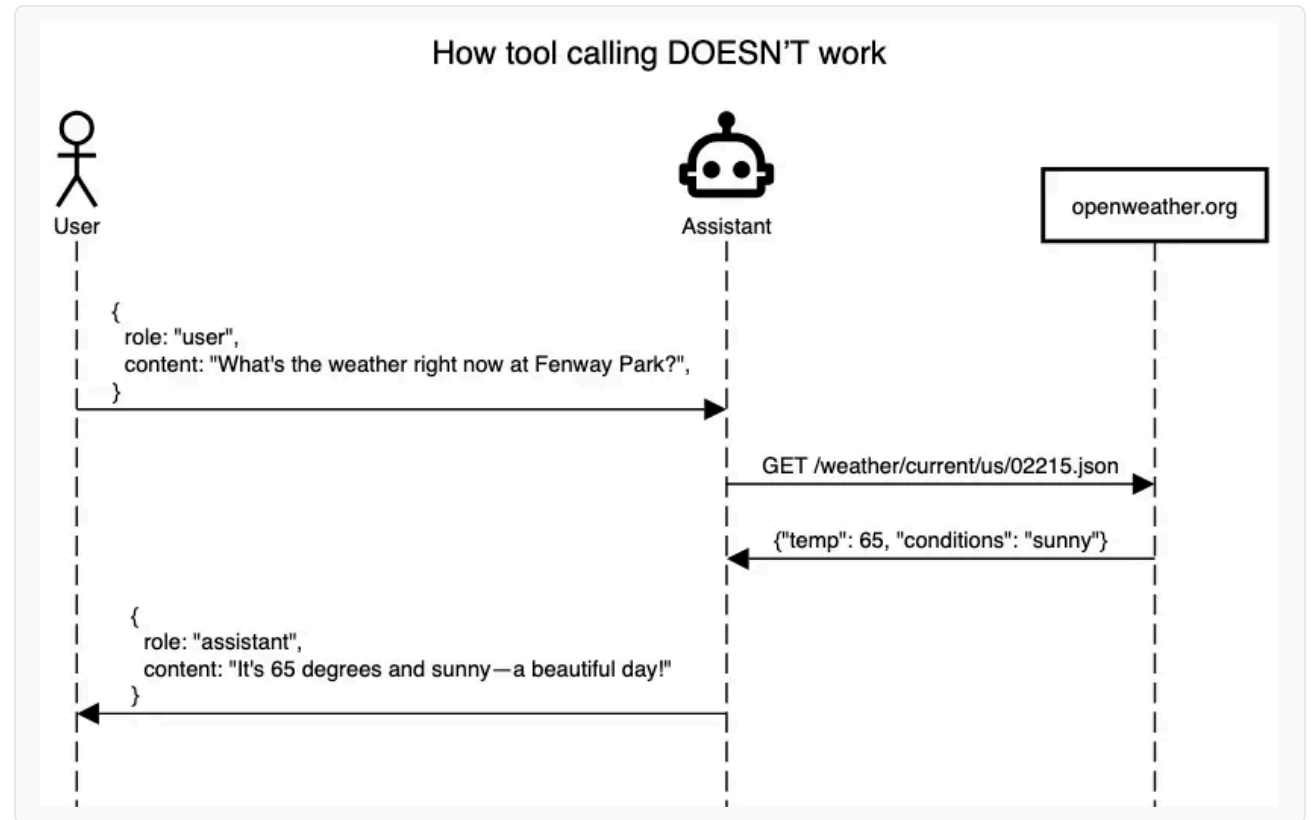
The model does not become a calculator, browser, or database. The surrounding software exposes those abilities and decides whether and how the requested action is executed.



The LLM Does Not Directly Call The API

Without software, a tool call is just text or structured intent.

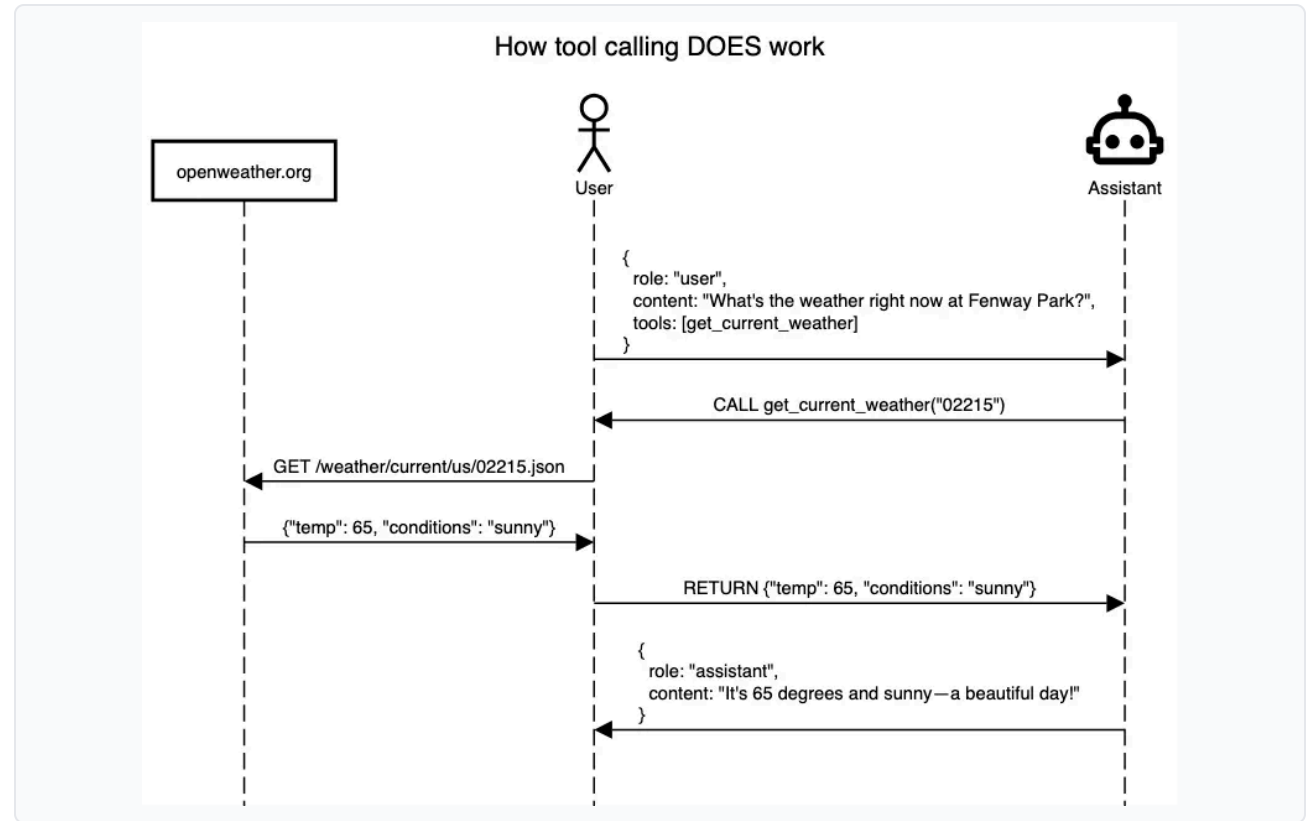
- the model predicts a tool-call-shaped message
- the host application parses it
- the host decides whether execution is allowed
- the host runs the code/API call and returns results



The Software Layer Executes The Call

The model proposes. The software disposes.

That software layer is where permissions, credentials, schemas, retries, logs, and safety checks live.



TOOLS

Demo: Sampling Allocation Tool

The user asks for a fieldwork allocation. The model should not invent arithmetic.

- call a deterministic allocation function
- pass structured arguments
- receive totals and validation checks
- explain the result in plain language

```
allocate_sample(  
    n_total = 1200,  
    strata = regions,  
    min_per_stratum = 50  
)
```

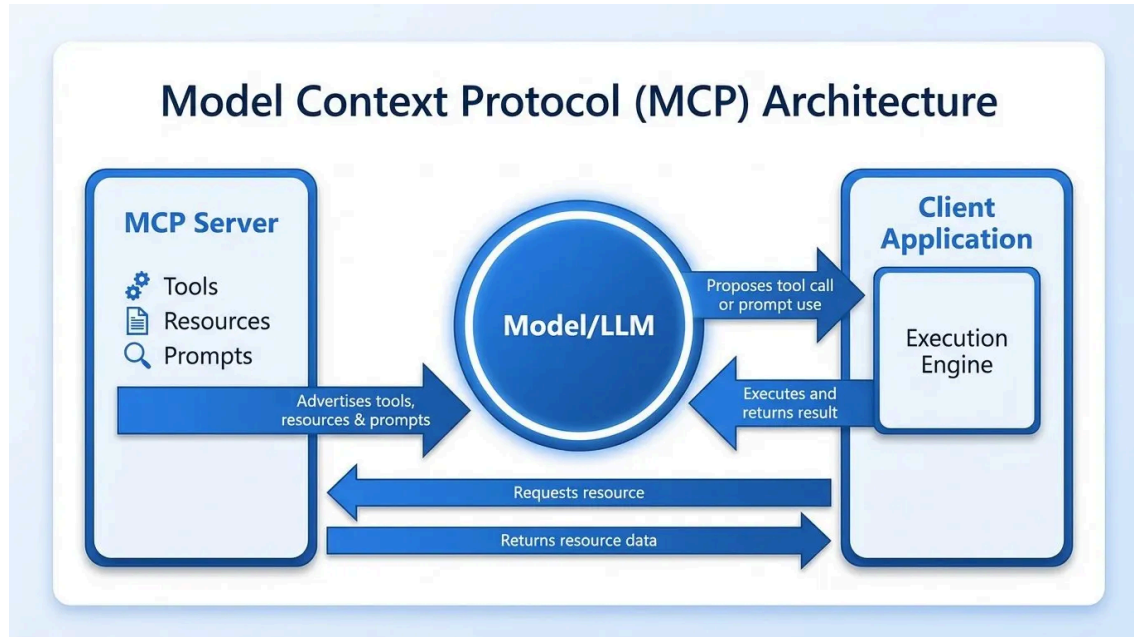
Give the LLM access to an interpreter and it won't rely on parametric memory.

Programmatic Tool Calling

Research control	Question for the methods section
Tool exposure	Which tools were available to the model?
Schema	What argument types and constraints were enforced?
Execution policy	Was tool use optional, required, or human-approved?
Permissions	What credentials and data scopes were available?
Trace	Were tool calls, arguments, outputs, and failures logged?
Validation	Were outputs checked before being returned to the model?

MCP: Standardising Tools for Models

Expose tools via servers; the model proposes, the client executes and returns results



Architecture

Server lists tools/resources → model suggests call → client runs and returns output.



What MCP is

An open standard to expose tools and resources to LLMs in a structured, reusable way.



Execution flow

Server advertises tools with names, descriptions, and schemas.

Model proposes a tool call based on those schemas.

Client app executes under constraints and returns results to the model.



Why it matters

Separation of concerns: model suggests actions; client enforces policy and safety.

Reusability: the same MCP server works across clients and models.

Auditability: structured calls/results are easy to log and reproduce.



Examples

Web search/scraping, code execution, file/project access, database/API queries via MCP servers.

MCP Standardizes The Software Boundary

MCP SERVER

- advertises tools, resources, and prompts
- wraps APIs, files, databases, or local code
- hides implementation details behind a stable interface

MCP



MCP CLIENT

- connects to one or more servers
- shows available tools to the model
- executes approved calls and returns results

MCP does not make the model act directly. It standardizes how software exposes actions and data to LLM clients.

MCP Flow



The important actor is the client software. It detects the tool call and runs the content of that call.

Architecture Example

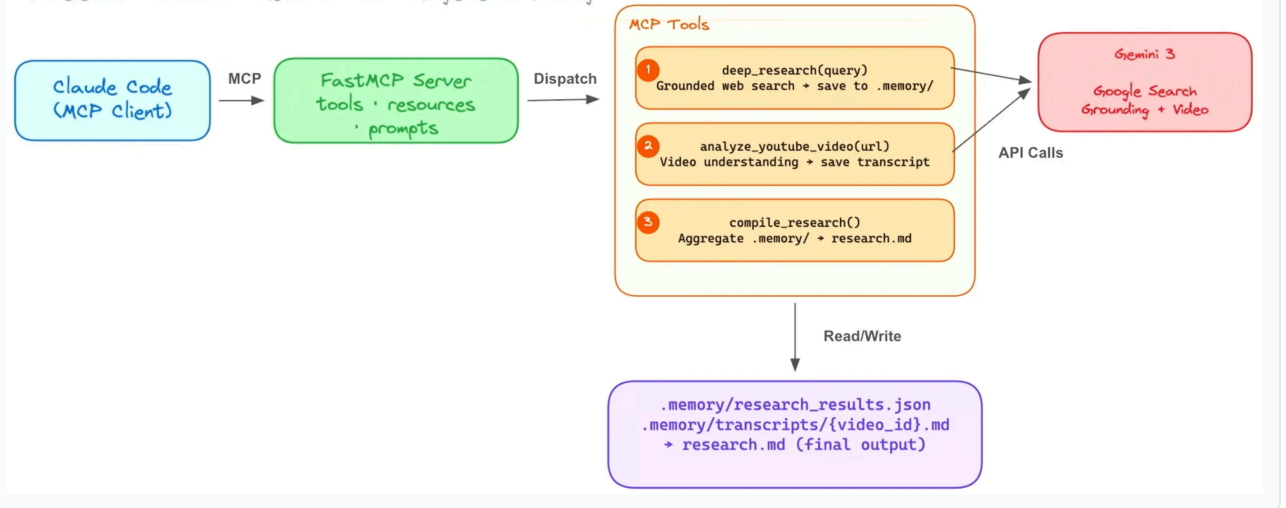
An MCP server can expose a small set of task-specific tools while hiding implementation details.

- tools = actions the agent can request
- resources = data the client can read
- prompts = reusable instruction templates

Architecture: How It All Connects

Deep Research Agent — MCP Server Flow

src/research/ • FastMCP • Gemini 2.5 Flash • Google Search Grounding



Demo: OpenAlex Literature Search

OpenAlex is the MCP demo: use a tool-backed scholarly index instead of asking the model to remember citations.

- query OpenAlex with explicit parameters
- return title, DOI, OpenAlex ID, year, authors, and abstract snippet
- ask the model to summarize only returned records
- verify citations against the tool output

```
search_openalex(  
    query = "LLMs survey research coding",  
    from_publication_date = "2020-01-01",  
    per_page = 5  
)
```

Jina: Webpage to Markdown

Turn noisy HTML into clean, LLM-friendly Markdown

GPT-4o & GPT-4o mini

GPT-3.5 & GPT-4

GPT-3 (Legacy)

Company-Privacy-Policy)

- * [Cookie Policy](https://www.nytimes.com/privacy/cookie-policy)
- * [Terms of Service](https://help.nytimes.com/hc/en-us/articles/115014893428-Terms-of-Service)
- * [Terms of Sale](https://help.nytimes.com/hc/en-us/articles/115014893968-Terms-of-Sale)
- * [Site Map](https://www.nytimes.com/sitemap/)
- * [Canada](https://www.nytimes.com/ca/)
- * [International](https://www.nytimes.com/international/)
- * [Help](https://help.nytimes.com/hc/en-us)
- * [Subscriptions](https://www.nytimes.com/subscription?campaignId=37WXW)
- * [Your Privacy Choices](https://www.nytimes.com/privacy/your-privacy-choices)

Clear

Show example

Tokens Characters

24,412 81989

Title: The New York Times – Breaking News, US News, World News and Videos

URL Source: https://www.nytimes.com/

Published Time: Wed, 03 Dec 2025 21:15:49 GMT

Markdown Content:

The New York Times – Breaking News, US News, World News and Videos

=====

[Skip to content](https://www.nytimes.com/#site-content)[Skip to site index](https://www.nytimes.com/#site-index)

GPT-4o & GPT-4o mini

GPT-3.5 & GPT-4

GPT-3 (Legacy)

```
href="https://static01.nytimes.com/newsgraphics/weather-hp-strip/8c1d23f2-1a67-4619-8f34-f86f9cd0d6b8/_assets/_app/immutable/chunks/weather-hp-modules.7d511448.js"><link rel="modulepreload" href="https://static01.nytimes.com/newsgraphics/weather-hp-strip/8c1d23f2-1a67-4619-8f34-f86f9cd0d6b8/_assets/_app/immutable/entry/app.b004294b.js"><link rel="modulepreload" href="https://static01.nytimes.com/newsgraphics/weather-hp-strip/8c1d23f2-1a67-4619-8f34-f86f9cd0d6b8/_assets/_app/immutable/nodes/0.f9113713.js"><link rel="modulepreload" href="https://static01.nytimes.com/newsgraphics/weather-hp-strip/8c1d23f2-1a67-4619-8f34-f86f9cd0d6b8/_assets/_app/immutable/nodes/3.3758f416.js"><!-- end hoisted interactive js--></body></html>
```

Clear

Show example

Tokens Characters

455,251 1311189

```
<!DOCTYPE html>
<html lang="en" class=" nytapp-vi-homepage " data-nyt-compute-assignment="fallback" xmlns:og="http://opengraphprotocol.org/schema/">
  <head>
    <meta charset="utf-8" />
    <title data-rh="true">The New York Times – Breaking News, US News, World News and Videos</title>
    <meta data-rh="true" name="description" content="Live news, investigations, opinion, photos and video by the journalists of The New York Times from more than 150 countries around the world. Subscribe for
```


I From Handicaps to Levers

🚫 Our handicaps:

Unknown training data (especially for closed models).

Alignment & RLHF we did not choose.

Hidden system prompts and safety layers.

⚙️ Our levers:

Context design:

What instructions and data we send.

Knowledge bases & RAG:

What corpus we give models to reason over.

Tools & MCPs:

What actions they can take, and where.

Sandboxes & IDEs:

How they interact with code and data.

Evaluation:

How we measure performance and stability.

Key takeaway: the user is not interacting with "a model"; they are interacting with the software stack wrapped around a model it.