Chapter 3 : Conditionning AI Agents behavior : goal, memory and context

An AI agent is often described, in its simplest form, as a looping LLM with tools. The model is prompted, produces a partial answer, may call an external tool, integrates the result, and then resumes the cycle, until a final answer emerges. At first glance this looks straightforward: a generative model wrapped in a loop. Yet as soon as we look more closely at what actually happens inside this loop, we encounter a profound and dynamic problem.

The problem is not the abstract power of large language models; it is the fragility of the context they rely on. Over the past two years, we have moved from 8k to 100k tokens of available context, an extraordinary technical leap, but one that comes with paradoxical effects: dilution of meaning, accumulation of noise, and a subtle loss of focus. Each new generation does not simply extend a clean logical chain ; it incorporates fragments of many different types: user queries, system prompts, tool outputs, retrieved documents, and fragments of earlier exchanges. The agent, in practice, is an evolving mixture of signals rather than a clear and understandable reasoning process.

Why does this matter? **Because an LLM is an autoregressive generative model. What it produces at every step is a function of its inputs**. And those inputs are themselves the product of earlier outputs, transformed and re-inserted in**to the loop. In this sense, the "state" of an agent is nothing other than the information transmitted through successive generations. If that information is noisy, misaligned, or unstructured, the next output will drift accordingly.**

This becomes especially critical in knowledge-intensive tasks, such as retrieval-augmented generation (RAG). Here, the model must seamlessly integrate external knowledge into its stream of consciousness. Tools, APIs, and retrievers are not luxuries, they are mechanisms to complete the agent's knowledge. But their contribution is only as good as the way their outputs are framed, filtered, and injected into the next turn of the loop.

Thus the key question of this chapter: What levers can we use on the input of each generation to improve the output of the next? How should we structure information? How do we reduce noise and preserve signal? How do we align the agent's evolving context with its intended goal?

One **productive way to see this is to treat the agent as a probabilistic stream of consciousness**. The generative function is no longer dependent only on the parameters of the model and the previous text. It also depends on a set of context-transformation functions mechanisms applied before generation: summarization, retrieval, ranking, filtering, role-based structuring, memory compression. These functions are not accessories. They are an essential part of the architecture of agency.

The ambition of this chapter is to bring together techniques that may appear disparate—system prompts, memory systems, retrieval pipelines, summarizers—and to show that they all revolve around a single dynamic problematic: how to shape the context so that the agent's probabilistic unfolding remains coherent, goal-aligned, and useful.

For now, we will mainly focus on « simple Agents », that is to say ReAct Agents, a LLM with a given set of tools provided that the same comments and reflexions will also be relevant for multi agent system (adding even more complexities).

I) Part 1 : Goals and Alignment in Agentic Generation : shaping the generative process

a) What is the point ?

At the most fundamental level, the **goal of an LLM** is deceptively simple: **optimize next-token prediction**. The model has been trained to estimate, with ever finer precision, the probability distribution of what token should follow given a sequence. This statistical imperative is the ground truth of its behavior.

Yet on top of this fundamental drive sits a **hidden goal**: **alignment with human expectations**. During pretraining and alignment phases, the model absorbs not just raw language patterns but also implicit objectives, safety, politeness, factuality, usefulness. These objectives are encoded in its parameters, not made explicit to the user. This is why they are "hidden": they live inside the statistical texture of the model, guiding it in ways that are difficult to fully observe or control.

A key question arises: **how do these goals interfere and coexist during generation?**

- On one side, the base model seeks to maximize likelihood.
- On the other, the aligned model must refuse harmful completions, avoid abusive content, or redirect unsafe queries.

In practice, alignment mechanisms are robust enough that a simple system prompt is usually not sufficient to override them. This tension between raw generative drive and layered alignment constraints is one of the central topics in the study of AI alignment.

But AI agents go further. They are not just generative models, they are **goal-seeking systems**, expected to plan and act in multi-step environments. Reasoning mechanisms like **chain-of-thought (CoT)** introduce intermediate goals: breaking down a user request into subproblems, solving them sequentially, and integrating results. This resonates strongly with the **previous generation of autonomous agents in reinforcement learning (RL)**. In RL, an agent was trained to orient its actions in order to maximize reward. The action space, however, was limited and discrete: move left, pick up an object, fire a laser. By contrast, LLM-based agents inhabit an **action space made of langage,** they produce content, instructions, and queries, all of which may themselves become new inputs. Instruction and possible actions (possible thanks to the function calling mechanism) are conditionned by the generation of a certain string of tokens.

This shift brings two challenges:

1. **Continuous and vast action space**: not just a handful of discrete actions but billions of possible sequences of text.
2. **Discretization problem**: choosing the “right” action toward a goal is equivalent to selecting the next token from a probability distribution. Goal-directed reasoning is collapsed into the mechanics of next-token prediction.

Thus, one of the main research questions of AI alignment is: **how can these different layers of goals, fundamental, hidden, emergent be balanced so that an agent can achieve its intended task?** In this chapter, we will not attempt to solve alignment at large, but rather focus on the **practical strategies** available to guide an AI agent toward accomplishing its mission.

« AI agents achieve goals by reasoning: systematically processing information, evaluating options, and selecting actions that maximize the likelihood of success. They rely on techniques such as **goal stacking** (ordering tasks by dependency) or **Monte Carlo sampling** (exploring possible continuations to approximate best outcomes). These strategies are not fundamentally different from those of RL agents; they extend them into the broader, messier, and far richer landscape of natural language.

But how to bridge this gap between words and goals ? From now on, we’ll define the goal as the main mission or task for which an AI Agent or an agentic system has been coded. If we classify LLM based agents according to how they natively interact with their environnement, most of them can be considered **goal-oriented** agents, that run only when triggered externally and who naturally stop their execution once the goal is fulfilled. This goal, shaped at the heart of the agent’s architecture (we’ll see how!) can possibly come in contradiction to some of the underlying goal(s)[1] persued by the LLM(s) that the agent uses.

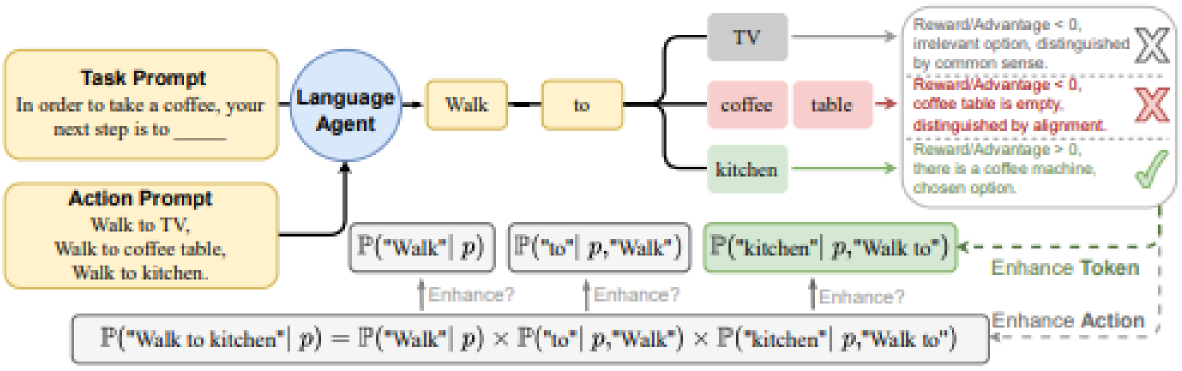
One of the techniques that are the most widely advocated to improve LLM Agent goal alignment is Reinforcement Learning. To sum things up, it is a training paradigm that leverage a mathematical function predicting the Reward of an outcome to update the model’s parameter, and it has been widely used during model alignment. While LLM Fine tuning is data intensive, Reinforcment learning usually requires less data for much better outcome. To be able to use **Reinforcment Learning** paradigm with autoregressive models, researchers had to find a way to descretize the textual data into an explicit reward function. During LLM alignment, reinforcement learning methods (like **RLHF**) needed to transform broad, fuzzy objectives (“be helpful, harmless, honest”) into something the model could optimize. The trick was to **discretize these objectives into per-token decisions**:

1. **Collect preferences** between whole model outputs (A vs. B).
2. **Train a reward model** to map a sequence of tokens → scalar score.
3. **Decompose this score back into the autoregressive loop**, so each next-token prediction gets a gradient signal that nudges it toward outputs humans preferred.

1 https://www.lesswrong.com/posts/nHDhst47yzDCpGstx/seven-sources-of-goals-in-llm-agents#fn14xc7jrjkw6h I really enjoyed the perspective of that blog

This way, high-level goals (alignment to human values) were reduced to the same unit the model already optimizes: **next-token prediction**, but now under a shaped reward instead of raw likelihood. So it was possible to **discretize the learning objective (next token prediction) into efficient Reward functions.** But it came at some cost. Preference corpora are hard to obtain, do not correspond to how people learn (human learn by contrast, but not by binary contrast), and it could introduce some biases depending on the properties of the prefered answers. But is it possible to improve the other way around and improve LLM capability at selecting tasks that would optimize goals oriented Reward functions ?

The emergent reasoning and planning capabilities of LLMs made them good candidates to power Action based agents without specific training (or surpassing their capabilities). The previous generation of task based RL agents operated in a **limited, discrete action space,** a handful of moves or commands. In contrast, language models inhabit a vastly richer space: their "actions" are **sequences of tokens**, with billions of possible trajectories. Every choice of a word, every punctuation mark, is an action in this space.But here lies the difficulty: if we want to use reinforcement learning to align a language agent toward a goal, how should we treat its actions? As entire sequences (a paragraph, a line of code), or as the tokens that compose them? Treating the whole sequence as one atomic action makes the **action space combinatorial**, exploding in size as soon as sequences become long. Yet treating each token independently risks destroying coherence: the model might optimize locally for tokens without regard for the meaning of the whole.


Task Prompt
In order to take a coffee, your next step is to ____
Action Prompt
Walk to TV,
Walk to coffee table,
Walk to kitchen.
Language Agent
Walk
to
TV
coffee
table
kitchen
Reward/Advantage < 0, irrelevant option, distinguished by common sense.
Reward/Advantage < 0, coffee table is empty, distinguished by alignment.
Reward/Advantage > 0, there is a coffee machine, chosen option.
ℙ("Walk"| p)
ℙ("to"| p,"Walk")
ℙ("kitchen"| p,"Walk to")
Enhance Token
Enhance?
Enhance?
Enhance?
Enhance Action
ℙ("Walk to kitchen"| p) = ℙ("Walk"| p) × ℙ("to"| p,"Walk") × ℙ("kitchen"| p,"Walk to")

This is precisely the tension addressed by a recent line of work, including the paper *"Reinforcing Language Agents via Policy Optimization with Action Decomposition (POAD)*[2]*"* which aim at incorporating action based objectives into the model's parameter in a meaningful way. The authors start from the observation that **credit assignment** is the central problem. When a language agent receives a reward for a completed sequence, how do we know which tokens contributed positively, and which led to failure? Optimizing at the sequence level hides the internal structure of the action; optimizing at the token level introduces bias, because not all tokens are equally responsible.

Their proposal, **Policy Optimization with Action Decomposition (POAD)**, introduces a middle way. Instead of ignoring the token structure or collapsing everything into one big sequence,

2 https://arxiv.org/pdf/2405.15821

they use a novel backup rule (the “Bellman backup with Action Decomposition”) to distribute rewards **consistently across tokens**. In effect, the method acknowledges that an action is a sequence, but it still evaluates the contribution of each token in light of the whole. The consequence is twofold: optimization remains **tractable** (additive rather than exponential in sequence length), and credit assignment becomes **granular** enough to guide token-level behavior without losing the coherence of the full action.

POAD agents learn faster and more stably than baselines. In more realistic, open-ended tasks where the agent must generate and debug code. Thist illustrates that the discretization of the action space into tokens is not just a metaphor, it is a technical bottleneck. When we say that “choosing the right decision to get closer to the goal is tantamount to a next-token prediction,” we are not speaking loosely. We are pointing at the precise locus where goal alignment lives: **the probability distribution over the next token**. If alignment is to succeed, it must operate at this scale, since every higher-level goal (write code, explain history, summarize evidence) is ultimately decomposed into a sequence of token-level decisions. The POAD framework shows one way to reconcile these scales. By providing **token-level credit signals that remain consistent with sequence-level goals**, it demonstrates how hidden objectives (alignment constraints, task-specific rewards) can be injected directly into the generative stream. This connects back to our discussion of hidden goals: alignment is not only an external system prompt or a top-level instruction, but also a **distribution of pressures acting on every micro-step of generation**.

Language agents are not just “RL agents with more actions.” They are **probabilistic systems where every token is a decision point**, and where context, memory, and alignment mechanisms all serve one purpose, to structure the distribution from which the next token is drawn.

If alignment begins with discretizing broad objectives into token-level decisions, the next challenge i**s how to *elicit* and represent the agent’s goal so it can be maintained throughout generation.** This brings us to the question of goal elicitation: how do we surface, structure, and keep the goal active in an agent’s context?

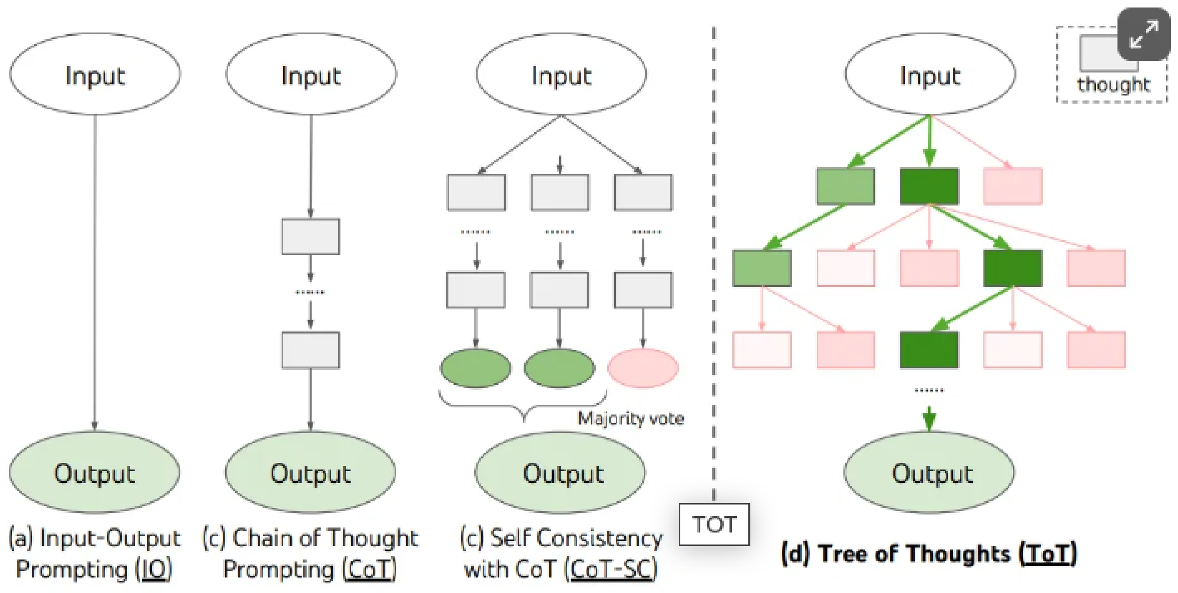
b) Goal elicitation techniques in Large Language Models

Prior to 2022, NLP machine learning models were typically trained for a specific task (e.g., classification, translation) or fine-tuned from a generic model pretrained on next-token prediction. With the advent of scaling laws and the rapid increase in model parameters, it was no longer necessary to fine-tune foundation models to perform many tasks. Instead, autoregressive models could be aligned to follow instructions—sometimes very complex ones—by leveraging their emergent capabilities. Solving a task thus becomes tantamount to generating a string (the action space, as we have seen, is embedded within this process) that satisfies a seed instruction. The shape of these instruction is therefore fundamental to ensure the goal’s completion.

The formulation of these instruction recieved a name on its own and became a kind of (esoteric) art : prompt engineering. Prompt Engineering is an experimental field which main goal is to study the best way to format and write LLM instruction to optimize their outputs on a given task.
It is highly experimental as LLM Generation depend on the model used, inference hyperparameters .. and as we ve seen instructions are just one part of the equation. An optimal

prompt for a model could yield poor results with another. This is why there is no certainty and reproductibility in the use of certain prompting techniques. The only thing that is possible is to use the scientific process and optimize generation by modifying one parameter at a time. If we fix both model and inference parameter, then in single turn setting, the prompt is the only thing we can tune. This process is called **prompt optimization** and can sometimes be automated. If the generative process is multiturn, then the content generated at each turn becomes a new and non deterministic component. Therefore prompt can only be optimized in an extrinsic fashion by evaluating the success on the general task.

If there is no magic trick or reproductible process to craft one prompt that would be the best for every model and settings (and this is why I do not call prompt engineering a science), there are however general principles and tendencies that regularly appear to improve model performance and are directions to explore while optimizing your prompt and some of them appear to be extremely relevant while building AI Agents instructions. Among the main strategies (I wont delve into details as its not difficult and many quality ressources are available) are :
* Direct elicitation
* Incorporating examples (one / few / many shot prompting) corresponding to the given task. Possible only with direct task not multi turn processes. Sometimes (weirdly enough) it can actually degrade performances compared to direct elicitation.
* Chain of Thought : asks the model to explicitely detail its reasoning and the different steps explored. It was proven very useful for reasoning and math tasks and appear to be directly applicable to AI agent planification. CoT can be seen as a mono turn process (ask the model to explicitely detail the different steps of its reasoning) or as a multi step process (parallel with turns of AI Agents) where only one step is created for each generation. This led to variants of the CoT paradigme where instead of just running the model once for each step, the model can be run multiple time (usually with a higher temperature) which will enable to select the most likely output at each turn, or explore many reasoning paths (and trying to select the best one via techniques like pruning, beam search, majority vote…). This idea of exploring the reasoning space can be seen graphicly and navigating through graph[3]. By shifting from the reasoning to the action space, these techniques are directly applicable to Agents.

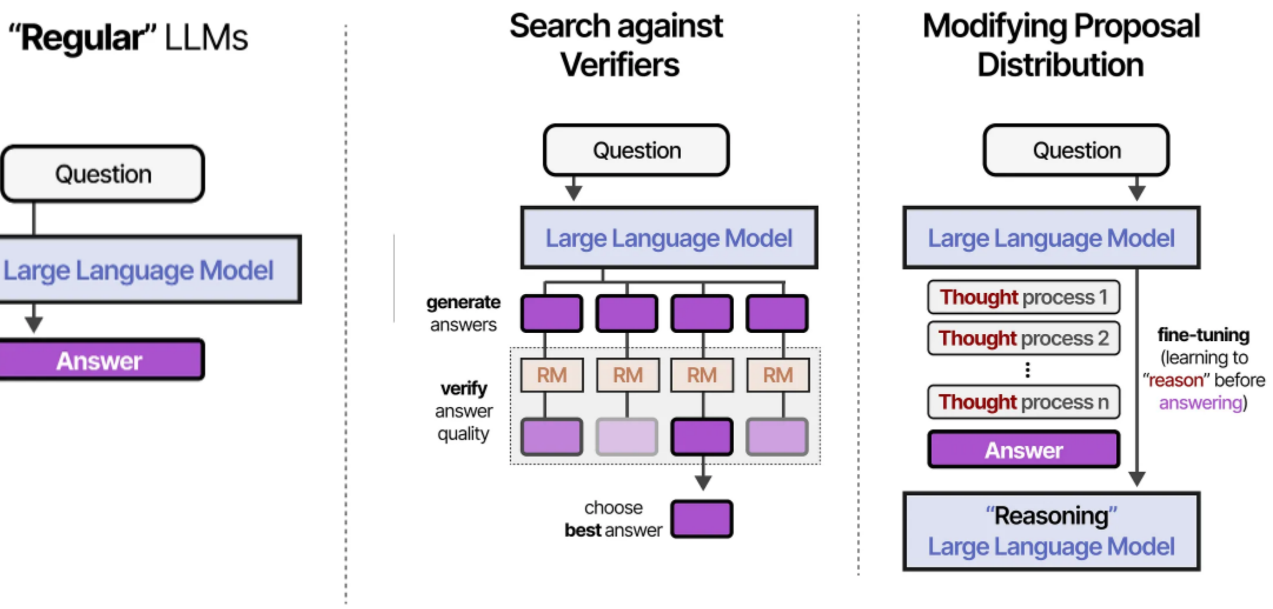


Input
Output
(a) Input-Output Prompting (IO)
(c) Chain of Thought Prompting (CoT)
Majority vote
(c) Self Consistency with CoT (CoT-SC)
TOT
thought
(d) Tree of Thoughts (ToT)

3 https://arxiv.org/abs/2305.10601

The same ideas are behind Time test scaling, the idea that launching smaller model on various « reasoning path » is more efficient than scaling model size and launching a big model just once[4]. Even if this family of techniques is very powerful, they show two major limitations.
1) they are expensive as not 1 but a combinatorial number of generations must be launched for just moving forward to the next turn (simple CoT is linear in number of calls, ToT could be quadratic…)
2) Prompts are static and cannot be adapted to the provided parameters. The issue is that while prompts might be good for some examples, they can fail with edge cases. For instance in a classification task, more guidance could be required to separate examples that are at the frontier between two classes. Even with few shot learning, the limited context size does not allow to stuff enough information into the prompt to cover and optimize every case.

In order to limit this generation combinatorial, LLM providers tried to teach model to natively « perform CoT inside **a single** generation turn », which means thoughts will be generated one after the other until the final answer is generated.[5]



There are two ways to modify inference ; horizontal by generating several options and vertical by modifying the proposal through the generation of multiple outputs. The technical difficulty is to achieve distillating reasoning into base models. It can be done via Supervised Fine Tuning on synthetic reasoning datasets, but Reinforcment Learning were key in achieving greater performances. First step to build reliable reasoning models was to separate reasoning content from the final answer. It was done with specific tags (cf I. c)) which enabled. The RL algorithm used in this process is called Group Relative Policy Optimization (GRPO) and is convieved to update the likelyhood of each step that led to a correct of false result. This specific alignment step was used to transform a basic model into a reasoning model. No examples were given on how the <think> process should look like and the RL process is based on the accuracy only. The model learnt by itself during each training round to generate more and more tokens and improve the likelyhood of correct answer. Reasoning models can be seen as onther hint of the shift from train time to test time compute. Using such models could improve planification and goal alignment in agents.

4 https://arxiv.org/abs/2408.03314
5 https://newsletter.maartengrootendorst.com/p/a-visual-guide-to-reasoning-llms my love of Marteens work is infinite

To solve thes second issue, the idea could be to dynamically update the prompt using the generation parameters.[6]A new paradigm for task alignment is **in-context learning (ICL)**. As surveyed in recent work, ICL refers to the ability of LLMs to "make predictions based on contexts augmented with a few examples." The central idea is that models can **learn from analogy**: given a handful of demonstrations in natural language templates, they generalize to new queries without any parameter updates. This is why ICL is often described as *few-shot prompting*.

If we transpose this paradigm to the world of AI agents, however, we encounter a challenge: **there are often too few examples** available to guide behavior. In classical ICL, a prompt is constructed with demonstration examples formatted in natural language templates, followed by the query. The LLM then treats the entire concatenated sequence as input and produces the prediction. This bears some similarity to agentic runs: tasks are learned without parameter tuning, templates act as vehicles for human knowledge, and the system adapts dynamically from contextual cues. The idea behind In Context Learning is to select the few shot examples from a vast collection of examples to only select the examples that are the most similar to the input data. This requires to already have some seed data at hand, but is done without training. For instance for a classification task one could reuse a dataset previously used to train a neural network, embed the examples in a vector store and only select the 3 most relevant (= similar to the new input data) annotated pairs to be stuffed in the prompt.

But the analogy has limits. Agentic runs typically lack rich demonstrations to seed the process, making them more brittle. Moreover, as with ICL, performance is highly **sensitive to the prompt template**—to the exact selection, ordering, and formulation of the constituents. Conciseness of demonstrations also matters, since longer prompts not only risk introducing noise but also raise **computational efficiency concerns**. Thus, both ICL and agentic prompting face the same design tension: how to encode just enough structured guidance into the context to elicit the desired behavior, without overwhelming or distorting the model's generative process.

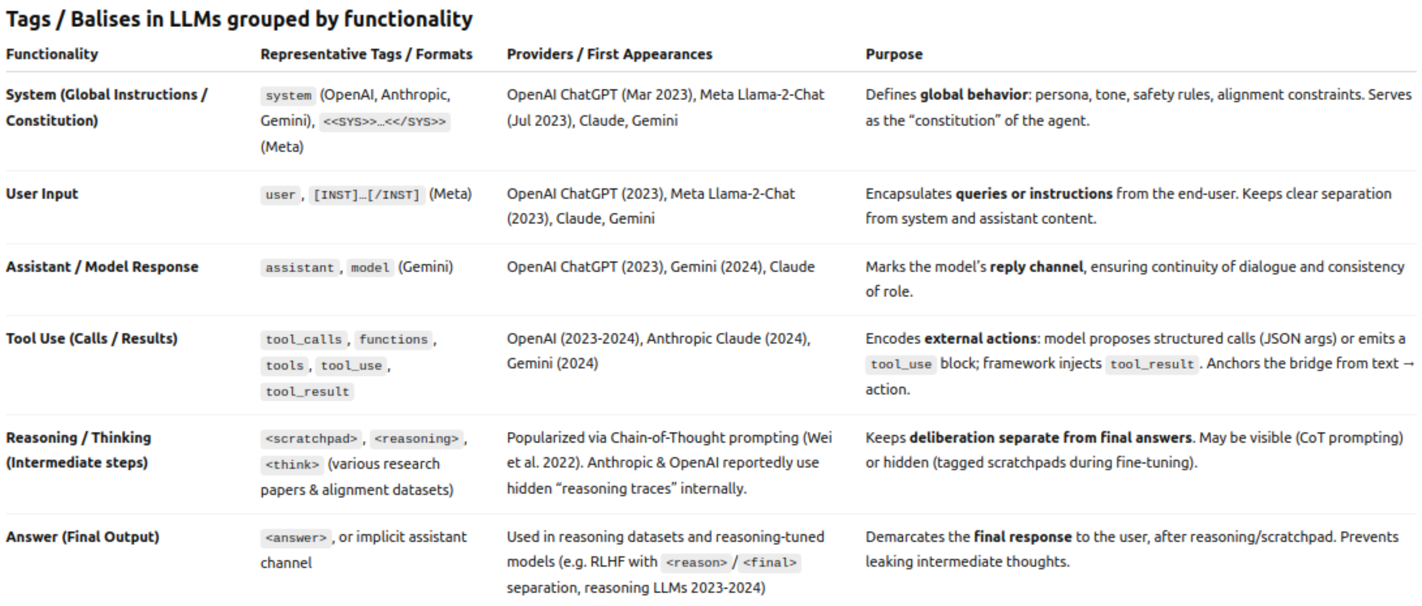
I.c Generation Tuning and the Architecture of Alignment

Goal alignment in AI agents is not only a matter of how we design prompts or structure plans at inference time. It is also profoundly conditioned by the **alignment phase of the model itself**. Long before an LLM becomes the motor of an agent, its generative tendencies have been tuned by supervised fine-tuning, reinforcement learning from human or AI feedback, and carefully designed instruction datasets. These processes leave deep imprints on the way the model handles goals, instructions, and reasoning. In other words, an agent does not start from a neutral generative baseline: it inherits an *architecture of alignment* that was already shaped upstream.

One of the most striking artifacts of this alignment is the proliferation of **tags and formatting conventions**. In order to separate reasoning from answers, or tool calls from narrative content, models have been trained to recognize and generate **categories of tags** such as `<reasoning>...</reasoning>`, `<scratchpad>...</scratchpad>`, `<tool>...</tool>`. These tags act as **balises**: invisible scaffolds that give structure to the generative process. During alignment, models learn that reasoning belongs in one space, answers in

6 If we denote llm the generative model, h the inference hyperparameters, p the prompt and a the arguments that fille the prompt template, we had response = llm(p(a), h)
Now we can define an prompt optimization function o and have response = llm(o(p, a)(a), h)

another, and that special markers control the transition from thought to action. While end-users of AI frameworks often remain unaware of these conventions, they are decisive for agents. They allow the model to maintain coherence across turns, to avoid leaking internal reasoning into external answers, and to separate intermediate planning from final outputs. In multi-turn scenarios, such scaffolding becomes indispensable to keep the agent oriented toward its goal. One has to carefully follow the tag use otherwise model generation will be chaotic.



Tags / Balises in LLMs grouped by functionality

Functionality	Representative Tags / Formats	Providers / First Appearances	Purpose
System (Global Instructions / Constitution)	`system` (OpenAI, Anthropic, Gemini), `<<SYS>>…<</SYS>>` (Meta)	OpenAI ChatGPT (Mar 2023), Meta Llama-2-Chat (Jul 2023), Claude, Gemini	Defines **global behavior**: persona, tone, safety rules, alignment constraints. Serves as the "constitution" of the agent.
User Input	`user`, `[INST]…[/INST]` (Meta)	OpenAI ChatGPT (2023), Meta Llama-2-Chat (2023), Claude, Gemini	Encapsulates **queries or instructions** from the end-user. Keeps clear separation from system and assistant content.
Assistant / Model Response	`assistant`, `model` (Gemini)	OpenAI ChatGPT (2023), Gemini (2024), Claude	Marks the model's **reply channel**, ensuring continuity of dialogue and consistency of role.
Tool Use (Calls / Results)	`tool_calls`, `functions`, `tools`, `tool_use`, `tool_result`	OpenAI (2023-2024), Anthropic Claude (2024), Gemini (2024)	Encodes **external actions**: model proposes structured calls (JSON args) or emits a `tool_use` block; framework injects `tool_result`. Anchors the bridge from text → action.
Reasoning / Thinking (Intermediate steps)	`<scratchpad>`, `<reasoning>`, `<think>` (various research papers & alignment datasets)	Popularized via Chain-of-Thought prompting (Wei et al. 2022). Anthropic & OpenAI reportedly use hidden "reasoning traces" internally.	Keeps **deliberation separate from final answers**. May be visible (CoT prompting) or hidden (tagged scratchpads during fine-tuning).
Answer (Final Output)	`<answer>`, or implicit assistant channel	Used in reasoning datasets and reasoning-tuned models (e.g. RLHF with `<reason>` / `<final>` separation, reasoning LLMs 2023-2024)	Demarcates the **final response** to the user, after reasoning/scratchpad. Prevents leaking intermediate thoughts.

At the same time, the **expansion of context size** has changed the conditions of alignment. Moving from 8k to 100k tokens and beyond has given agents a much larger canvas on which to sustain their goals. Yet this expansion comes with paradoxical effects: the more information we can pack into the prompt, the more fragile the signal-to-noise ratio becomes. Long contexts carry not only goals and instructions, but also accumulated tool outputs, memories, user corrections, and digressions. Without careful management, the central mission can easily become buried under peripheral details. Thus, **goal alignment in practice depends not just on context length, but on context curation,**the ability to filter, compress, and prioritize information so that the agent's generative process remains anchored to what matters most.

Taken together, these factors highlight the **core problematic**: how to ensure that an agent's goal remains alive at the heart of the generative loop. Tags and formatting conventions provide structural anchors; alignment phases embed hidden preferences into the model's parameters; expanded contexts give room for richer histories but also risk diluting focus. Generation tuning, in this sense, is not a technical afterthought but the backbone of agentic behavior. The challenge for agent designers is to build memory systems, prompting strategies, and context-engineering pipelines that do not fight against these inherited structures, but rather **leverage them to continually re-ground the agent in its mission across turns**.

If tags and alignment formats provide the structural anchors that keep an agent oriented turn by turn, memory provides the substance that carries its goals and knowledge across turns. Memory management thus becomes a key question in Agent design.

II) Building memories for AI Agents

a) From AI Assistant to Agent Memory

A “classic” conversation with an LLM is structured as a list of messages categorized as **system** (the system prompt), **user** (the user’s message), or **assistant** (the LLM’s reply). As soon as we enrich the system prompt with tool descriptions and begin appending observations about tool use, the number of tokens consumed quickly grows. Since the context window of a model is not infinite, if an agent performs a long series of tool calls and reasoning steps, this window will eventually saturate.

This is not just a theoretical concern. Even in ordinary assistants like ChatGPT, if a conversation is long enough, it can no longer fit entirely into the context window. The model therefore only receives the last few turns as input for the next generation. This creates the illusion of memory, but in practice it often leads to problematic forgetting. In other words, both conversational assistants and more sophisticated agents face the same underlying issue: **how to keep relevant information accessible when the generative context is bounded**.

To mitigate these limitations, more elaborate **memory managers** have been devised. Their role is to decide which content should be included in the prompt window for the next generation. Common strategies include:

- **Buffer memory** (Conversation Buffer Window Memory): keeps only the last k messages in the conversation.
- **Summary memory** (Conversation Summary Memory): every 10 or 20 messages, a separate LLM summarizes the earlier part of the conversation, recursively if needed. This compresses the context effectively but at the cost of extra LLM calls and potential loss of fine-grained details.
- **Knowledge-graph memory**: each message is parsed and its content is added to a graph structure. This allows relational retrieval later, but introduces computational cost and reliability challenges.
- **Semantic index memory**: messages are embedded and stored in a vector index; at each turn, only the most relevant past messages are retrieved via semantic search to form the new context.

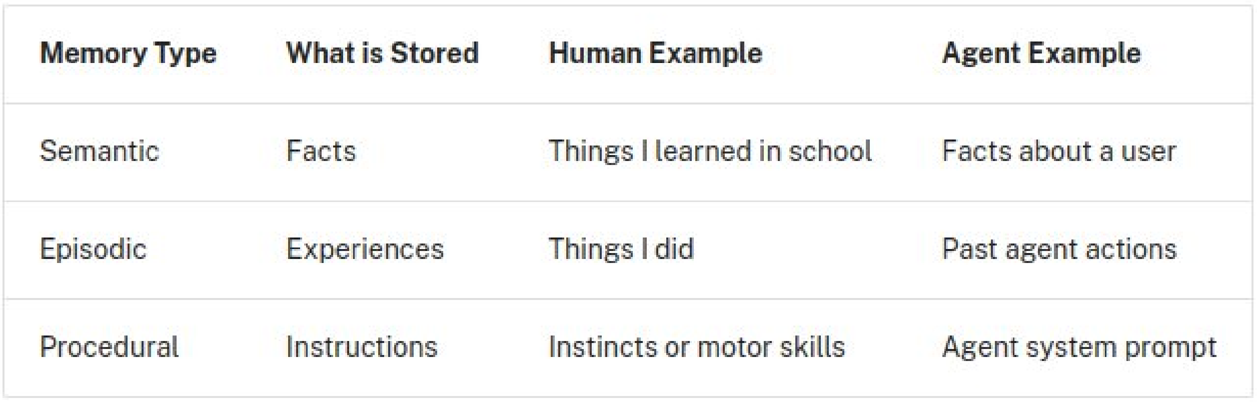
All these mechanisms can be seen as different forms of **conversation compression**, and they can also be transposed to **autonomous agents**. Agents need access not just to recent turns, but to a wide range of heterogeneous information: facts, preferences, past actions, external knowledge.

This is where theoretical work such as *Cognitive Architectures for Language Agents (CoALA)* [7]becomes particularly useful. Inspired by research on animal cognition, the authors distinguish three main types of memory:

- **Semantic memory**: knowledge of facts — general world knowledge, domain-specific expertise, or user preferences.
- **Episodic memory**: records of past experiences — in an agent, this means actions performed and their outcomes, essential for planning future steps.

7 https://arxiv.org/pdf/2309.02427

- **Procedural memory**: knowledge about how the agent itself functions — its overall mission, the tools it can use, its persona, typically encoded in the system prompt.



Memory Type	What is Stored	Human Example	Agent Example
Semantic	Facts	Things I learned in school	Facts about a user
Episodic	Experiences	Things I did	Past agent actions
Procedural	Instructions	Instincts or motor skills	Agent system prompt

The key argument is that if an agent needs all three types of information to accomplish its mission, then these must be stored and retrieved differently, using distinct mechanisms. This leads to the idea of cognitive agents: architectures capable of dynamically manipulating multiple memory structures in parallel.

he *CoALA* paper has been seminal in broadening reflections on agent memory and providing theoretical context to dozens of subsequent works tackling this complex topic. However, many frameworks have quickly adopted the terminology introduced in the paper—sometimes at the risk of implementing multiple types of memory for the sake of implementation itself, without seriously evaluating the benefits for agent performance. *LangGraph* seems to me the most emblematic example of this tendency. In a short course dedicated to agent memory[8] (one of the very rare resources explicitly on the subject), the CoALA terminology was even used as chapter titles to present some of the library’s components. Yet these implementations are not trivial, and the course neither explained nor assessed the real advantages of this new kind of memory. Most strikingly, no reflection on computational trade-offs was provided, leaving open the question of whether such architectural complexity is justified in practice.

Everyone agrees on what **short-term memory** means for an agent: essentially a filter over the recent messages of the conversation, the content that fits into the model’s context window. The real challenge lies in the design of **long-term memories**—how to represent, store, and retrieve the broader set of information accumulated during the agent’s execution. The central question, then, is how these long-term stores can be transformed into effective short-term context, so that the agent consistently stays aligned with its goals across turns.

b) inventing long term memories : suggested hypotheses and architecture

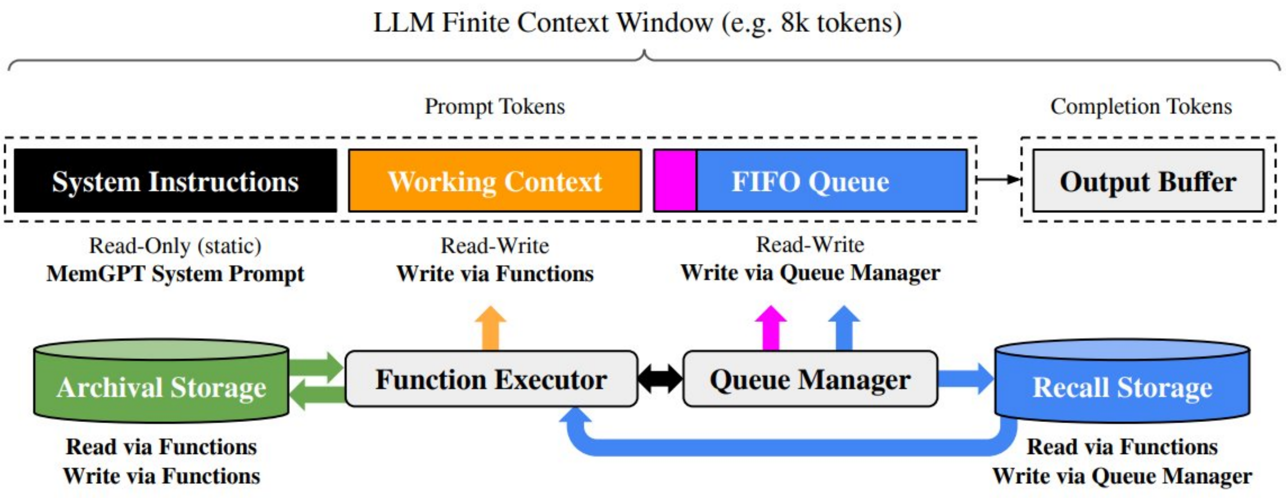
Across frameworks, short-term memory is solved uniformly (buffers/windows over recent turns), while long-term memory is still a design space, not a standard: most stacks push you toward

8 https://www.deeplearning.ai/short-courses/long-term-agentic-memory-with-langgraph/

retrieval-based memories (vector/KG, summaries) and externalized persistence (threads/checkpoints, memory banks, MCP servers). Google's Agent Builder is the most opinionated with a managed Memory Bank; LangGraph is the most explicit about state/resume mechanics; Strands and smolagents provide lightweight patterns and rely on pluggable stores. OpenAI's Swarm stays stateless (use Assistants API for hosted memory). What's missing almost everywhere are rigorous, comparable ablations on goal-retention vs. cost/latency—exactly the evaluation gap your course will address before surveying academic proposals. During the last six month, very interesting works have been published on AI Agents memory.

One of the first paper on agent memoy, *MemGPT*[1], takes a step further by providing a concrete **implementation of memory management mechanisms**. The context window for each new generation is composed of several distinct sources:

- **The system prompt**, which encodes the global function of the agent.
- **A queue of recent conversation turns** (FIFO), ensuring short-term continuity.
- **A set of essential facts about the user or the agent's persona**, stored in a key–value database. These "core memories" can be created or updated dynamically by the agent itself through dedicated memory-editing tools exposed via function calling. At each generation, the contents of this core memory are automatically injected into the system prompt, guaranteeing that the agent retains access to its most important long-term facts.
- **Archival memory**, an index of past interactions, which agents can query when they need to recall details from earlier execution steps.



If I am not mistaken, this was the first paper to place the **question of memory at the very heart of AI agent implementation**, at a time when most mainstream frameworks had not yet given it much attention. By enabling agents not only to access but also to **edit their own memory**, *MemGPT* introduced a genuinely agentic mechanism: a model can now decide which facts are worth storing or updating, and retrieve them as needed to extend its context window.

That said, the approach comes with limitations. The effectiveness of memory updates depends heavily on the **judgment of the LLM itself**—whether it correctly identifies which information deserves to enter the "core memory" and whether retrieval from archival memory is precise enough.

In other words, while *MemGPT* elegantly addresses context-window limitations, it also places a heavy burden on the generative model's ability to make consistent, reliable memory-management decisions. The key idea brought by this paper is that memory is not just a function, it is a complete software layer, that we can call **memory management, where elements are not only retrieved but also written, edited, replaced…** In MemGPT memory edition simply took the shape of writing in a dictionnary, but more advanced approached can be envisioned.

A-MEM (*Agentic Memory for LLM Agents*, Xu et al., 2025[9]) is one of the first frameworks to treat memory not simply as a container of past turns, but as a living structure that evolves over time. The proposal is inspired by the *Zettelkasten* method used in human note-taking: instead of just storing past content, each new memory is saved as an atomic note enriched with attributes (context, tags, keywords). These notes are then dynamically linked to existing ones based on similarity, and older memories can be revised in light of new evidence. The result is a network of interconnected memories, rather than a flat buffer or vector store.

The context window for the model is then composed not only of recent turns, but of curated, structured memories drawn from this evolving network. This allows the agent to bring into play not just what was said last, but what is semantically or episodically connected to the current situation.

From the perspective of CoALA's taxonomy, the structured notes and tags play the role of semantic memory (facts, contextual knowledge). The dynamic linking and updating mechanisms resemble episodic memory, since they capture and reinterpret past events as new ones occur. The procedures by which the agent creates, edits, and retrieves memories form part of its **procedural memory,** the "rules of use" that govern how memory participates in generation.

Placed against our central problematic, A-MEM highlights a crucial shift: memory is not just about fitting more tokens into context, but about **structuring information so that an agent's goals remain accessible across turns**. By deciding what to store, how to link it, and when to update it, A-MEM effectively shapes the inputs that condition every new generation, ensuring better goal alignment over long horizons. Of course, this complexity raises new questions. The gains reported are promising, but the framework introduces heavier computational costs (continuous tagging, linking, updating), and it is unclear how well such systems scale to noisy, real-world interaction histories. In the end, we can need up to 4 LLM calls for just one conversaton turn just to properly handle memory.

Academia has been much innovative. If we try we can recompose a general picture of the suggested implementations that are not implemnted yet in the most widely used frameworks.

Chronology of Memory Innovations in AI Agents (2019–2025)

*** 2019–2021 — Early retrieval and scratchpads.**
Before the "agent" terminology took hold, memory was mostly framed as **retrieval augmentation**: external databases or embeddings used to re-inject facts into prompts. Experimental work on *scratchpad mechanisms* (letting models write temporary notes) foreshadowed later interest in short-term working memory, but no agent-specific architectures existed yet.

9 https://arxiv.org/abs/2502.12110

*** 2022 — First non-parametric, persistent memory.**
With *MemPrompt*, researchers introduced a simple but powerful idea: store user corrections or failures in a **key–value store** and inject them into prompts during later runs. This was the first concrete demonstration that LLMs could benefit from a **persistent external memory** without retraining, seeding the now-standard notion of "long-term memory modules."

*** Early 2023 — Generative agents and reflection.**
The famous *Generative Agents* experiment showed characters in a sandbox town equipped with **episodic logs of observations**, a **reflection process** to summarize and synthesize them, and **retrieval** to condition future actions. This combination of *store → reflect → retrieve* made memory feel *agentic*: not just passive storage, but something that actively shaped behavior across time.

*** Mid–Late 2023 — Formal taxonomies and tiered memory.**

CoALA (*Cognitive Architectures for Language Agents*) provided a conceptual framework by distinguishing **semantic memory** (facts), **episodic memory** (experiences), and **procedural memory** (rules and instructions).

MemGPT operationalized this taxonomy in code: a **buffer** for recent turns, a **core key–value memory** that the agent could **edit via tool calls**, and an **archival index** for long-term retrieval. This was the first fully-fledged demonstration of **LLM-driven memory editing**, where the model itself decided what to store and update.

MemoryBank followed a similar route, emphasizing **continual long-term updates** of user information and preferences, bridging the gap between static assistants and adaptive agents.

*** 2024 — Stress-testing long horizons.**
The *LoCoMo* benchmark (300+ turns, multi-session) exposed the weaknesses of existing designs: even with longer context windows and retrieval, agents **forgot causal chains and temporal relations**. This shifted attention from "can we store more?" to "**how do we structure and select what matters?**"

*** 2025 — Structured, evolving, and secure memory.**

A-MEM proposed a Zettelkasten-inspired design: every new memory is an **atomic note with attributes (tags, keywords)** that is **linked to related notes** and can evolve as new information arrives. Memory becomes a **dynamic graph** rather than a flat buffer or vector index.

LongMemEval formalized the evaluation side, defining core abilities (information extraction, temporal reasoning, multi-session recall, knowledge updates, abstention) and a three-stage memory pipeline: **index → retrieve → read**.

Mem0 aimed for practicality, showing how **salience-based extraction + consolidation + retrieval** could yield **cheaper and faster memory** (90% token savings, reduced latency) while even experimenting with **graph-based memory**.

MINJA warned of vulnerabilities by showing how **memory injection attacks** could poison long-term stores, while *Memory-R1* explored **reinforcement learning for memory policies**—teaching agents *what to remember* and *when to forget*.

Activity: Exploring Cutting-Edge Proposals in Agent Memory (2025)

Lets (I mean let YOU) examine five other recent research papers that each introduce innovative approaches to agent memory. Your task is to analyze one system in depth each system to get a better understanding on how memory management is implemented. One paper among[10] :

1. **Mem0: Building Production-Ready AI Agents with Scalable Long-Term Memory**
2. **LongMemEval: Benchmarking Chat Assistants on Long-Term Interactive Memory**
3. **Reflective Memory Management (RMM): In Prospect and Retrospect**
4. **Zep: A Temporal Knowledge Graph Architecture for Agent Memory**
5. **R^{3}Mem: Bridging Memory Retention and Retrieval via Reversible Compression**

On the paper you selected, answer the following questions :

* what kind of memory is implemented ? Refer to the Coala taxonomy.

* Which NLP techniques are used ?

* Describe the memory retrieval and update process

* Evaluation methods and benchmarks in the article ?

* Are there solid elements on the cost/latency trade-offs ? What do you think about the proposal ?

Despite the creativity of these proposals, there is still no clear consensus on the “best” way to implement memory in AI agents. What emerges instead is a landscape of trade-offs. How can memory scale with time and conversation length without drowning the model in irrelevant detail? How much additional token cost is introduced by memory management systems that summarize, retrieve, or re-inject information, and is that cost justified by measurable gains? These questions are often overlooked, yet they are central: the true challenge is not only *what to remember*, but *how to transform memory into effective short-term context*.

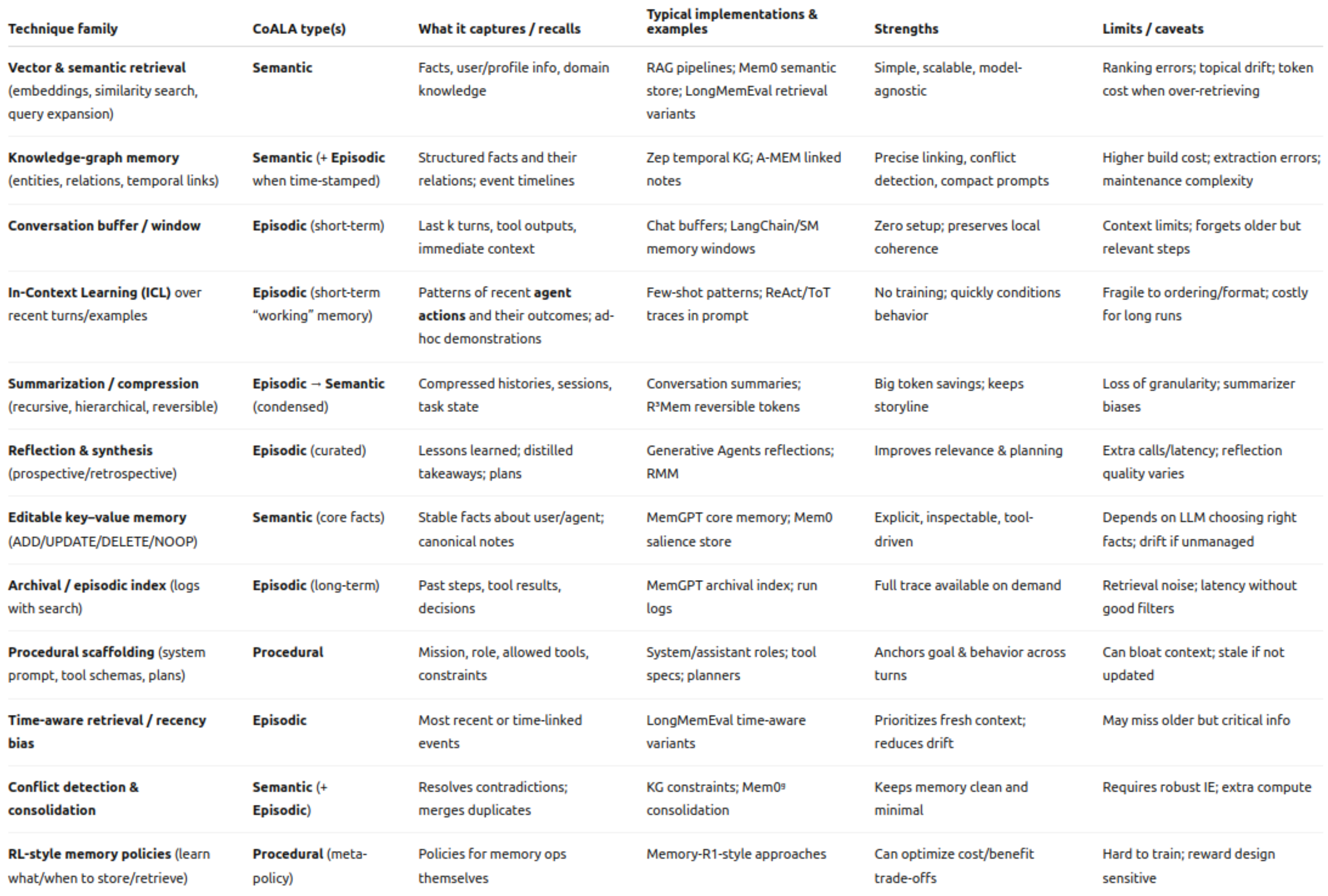
c) Implementing and evaluating long term memory

Designing memory for AI agents relies on a broad spectrum of NLP techniques: some approaches build knowledge-graph structures with extracted entities, relations, and temporal links; others embed past interactions into vector spaces for semantic retrieval and query expansion; still others compress conversation histories through recursive summarization, hierarchical abstraction, or even reversible compression. Beyond storage and retrieval, agents may edit their own memories by adding, updating, or deleting facts, consolidating salient details, or reflecting on past events before storing them, while reinforcement learning has even been used to train policies for when and what to remember. Memory thus emerges as a complex but fertile research space that mobilizes nearly all areas of NLP and offers a creative playground for experimenting with new agent designs[11].

10 There are more if you wanna have fun https://github.com/masamasa59/ai-agent-papers/blob/main/capability-papers/memory.md

11 See TP 3

The following table (maybe too exhaustive, I concede) assigns the NLP techniques used by the papers we gathered and connected to the CoALA taxonomy. We can deduct from this panoramix perspective that :



Technique family	CoALA type(s)	What it captures / recalls	Typical implementations & examples	Strengths	Limits / caveats
Vector & semantic retrieval (embeddings, similarity search, query expansion)	**Semantic**	Facts, user/profile info, domain knowledge	RAG pipelines; Mem0 semantic store; LongMemEval retrieval variants	Simple, scalable, model-agnostic	Ranking errors; topical drift; token cost when over-retrieving
Knowledge-graph memory (entities, relations, temporal links)	**Semantic** (+ **Episodic** when time-stamped)	Structured facts and their relations; event timelines	Zep temporal KG; A-MEM linked notes	Precise linking, conflict detection, compact prompts	Higher build cost; extraction errors; maintenance complexity
Conversation buffer / window	**Episodic** (short-term)	Last k turns, tool outputs, immediate context	Chat buffers; LangChain/SM memory windows	Zero setup; preserves local coherence	Context limits; forgets older but relevant steps
In-Context Learning (ICL) over recent turns/examples	**Episodic** (short-term "working" memory)	Patterns of recent **agent actions** and their outcomes; ad-hoc demonstrations	Few-shot patterns; ReAct/ToT traces in prompt	No training; quickly conditions behavior	Fragile to ordering/format; costly for long runs
Summarization / compression (recursive, hierarchical, reversible)	**Episodic → Semantic** (condensed)	Compressed histories, sessions, task state	Conversation summaries; R^3Mem reversible tokens	Big token savings; keeps storyline	Loss of granularity; summarizer biases
Reflection & synthesis (prospective/retrospective)	**Episodic** (curated)	Lessons learned; distilled takeaways; plans	Generative Agents reflections; RMM	Improves relevance & planning	Extra calls/latency; reflection quality varies
Editable key–value memory (ADD/UPDATE/DELETE/NOOP)	**Semantic** (core facts)	Stable facts about user/agent; canonical notes	MemGPT core memory; Mem0 salience store	Explicit, inspectable, tool-driven	Depends on LLM choosing right facts; drift if unmanaged
Archival / episodic index (logs with search)	**Episodic** (long-term)	Past steps, tool results, decisions	MemGPT archival index; run logs	Full trace available on demand	Retrieval noise; latency without good filters
Procedural scaffolding (system prompt, tool schemas, plans)	**Procedural**	Mission, role, allowed tools, constraints	System/assistant roles; tool specs; planners	Anchors goal & behavior across turns	Can bloat context; stale if not updated
Time-aware retrieval / recency bias	**Episodic**	Most recent or time-linked events	LongMemEval time-aware variants	Prioritizes fresh context; reduces drift	May miss older but critical info
Conflict detection & consolidation	**Semantic** (+ **Episodic**)	Resolves contradictions; merges duplicates	KG constraints; Mem0^g consolidation	Keeps memory clean and minimal	Requires robust IE; extra compute
RL-style memory policies (learn what/when to store/retrieve)	**Procedural** (meta-policy)	Policies for memory ops themselves	Memory-R1-style approaches	Can optimize cost/benefit trade-offs	Hard to train; reward design sensitive

* semantic memory has been the type of memory that have been the most widely explored
* agentic memory managemnt (used of tool use) are often used despite latency costs
* for episodic memory, In Context Learning and Reinforcement Learning are very promising
* The plebiscited techniques are text retrieval, entity extraction and writing elemnets (CRUD)…

Evaluating agent memory is particularly difficult because there is no single metric that captures what it means to "remember." Memory can fail by forgetting relevant details, by retrieving irrelevant noise, or by introducing contradictions, and each of these errors has different effects on an agent's behavior. Traditional NLP benchmarks (QA accuracy, perplexity) are poorly suited, so new datasets have been designed specifically to stress long-term memory. **LoCoMo** (2024) introduced very long, multi-session conversations to test whether models could recall facts and causal chains hundreds of turns later. **LongMemEval** (2025) refined this idea, defining abilities such as temporal reasoning, multi-session continuity, knowledge updates, and even abstention when memory is irrelevant. Together, these datasets show that while memory architectures can improve recall, trade-offs remain: gains often come at the cost of higher latency, extra token usage, or reduced granularity. This evaluation gap remains one of the hardest open problems in the field.

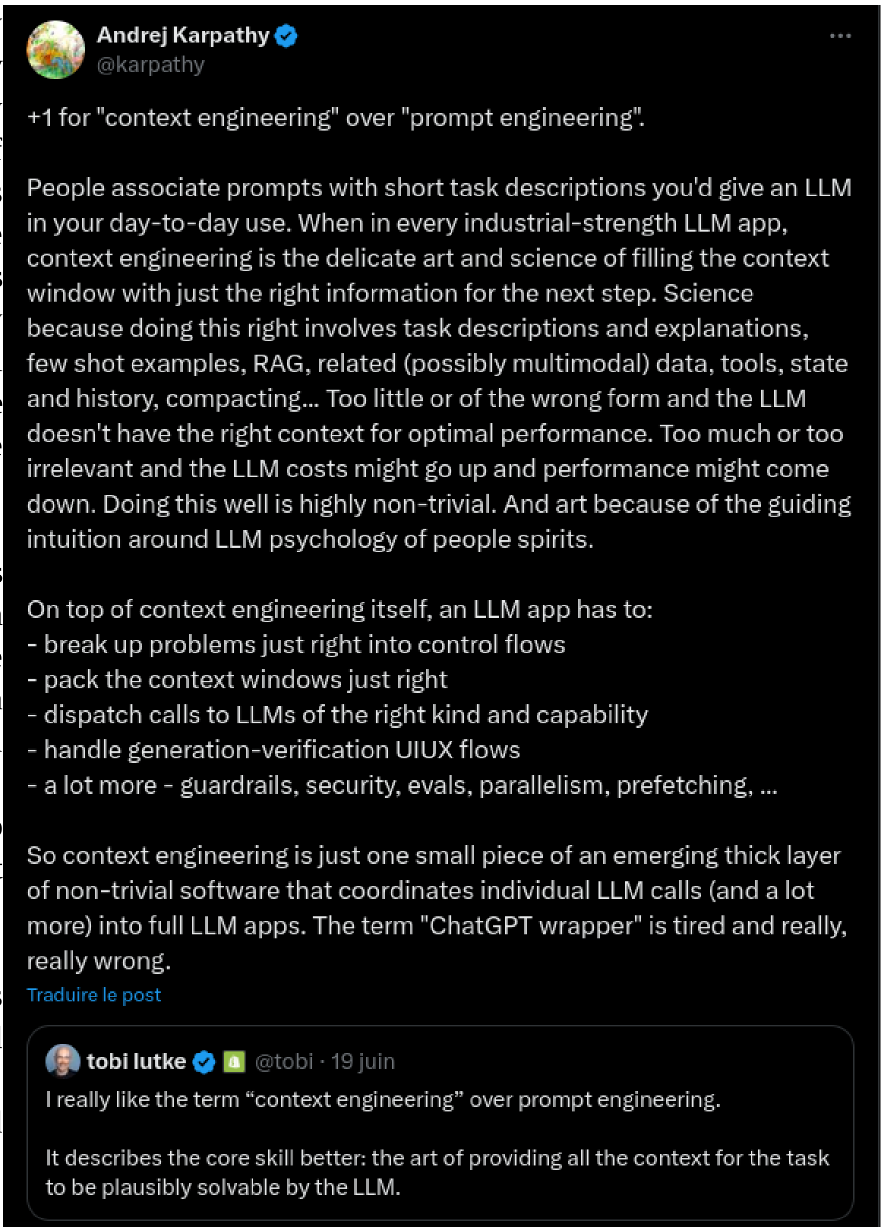
III) The Modern Art of Context Engineering

Even if Goal alignment and memory management seem to be seperate issue and leverage different techniques in their implementation, they all boil down to a single and rebarbative task : compilin the string that will be sent to an LLM to generate the next turn of conversation. This prompt not only contains the base instruction of the agent, but also potentially some elements coming from previous generations, informations retrieved from a semantic memory, recalls on where in the execution of the global task the AI Agent is so far… The string provided to the LLM for next turn generation in an AI Agent is therefore no longer only a prompt template filled with some variable. It a dynamic space filled with various components that c/should be activated and deactivated depending on the task and its execution state. Optimizing this filling operation, a new art coined « Context Engineering » becomes central in the design of AI Agents.

a) From Prompt to Context Engineering

The term was first briefly coined by Shopify CEO Tobias Lütke. It quickly gained traction ofter being approved by Andrej Karpathy, considered as one of the founding fathers of modern AI. This citation was indeed pronounced in the context of the emergence of AI Agents and the conclusion that they performance is not hinderd by technical difficulty, but by the « quality of the context provided » with regard to the limited context size available.

From this conclusion, Agents builders begun to extend their definition of what LLM context is beyond the mere prompting. In his commentary on Lütke's initial take, Karpathy insists on several ideas :
* the fact that prompting is associated to short tasks (and its implied is not enough anymore)
* Many technical skills are involved, making context engineering less accessible to non tech profile compared to prompt engineering
* has to do with performance and deployment costs
* it is hard (for now)



Andrej Karpathy
@karpathy

+1 for "context engineering" over "prompt engineering".

People associate prompts with short task descriptions you'd give an LLM in your day-to-day use. When in every industrial-strength LLM app, context engineering is the delicate art and science of filling the context window with just the right information for the next step. Science because doing this right involves task descriptions and explanations, few shot examples, RAG, related (possibly multimodal) data, tools, state and history, compacting... Too little or of the wrong form and the LLM doesn't have the right context for optimal performance. Too much or too irrelevant and the LLM costs might go up and performance might come down. Doing this well is highly non-trivial. And art because of the guiding intuition around LLM psychology of people spirits.

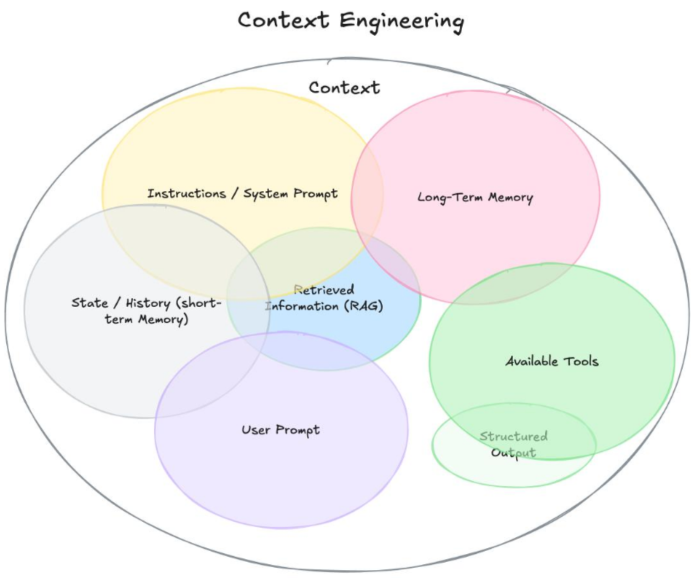
On top of context engineering itself, an LLM app has to:
- break up problems just right into control flows
- pack the context windows just right
- dispatch calls to LLMs of the right kind and capability
- handle generation-verification UIUX flows
- a lot more - guardrails, security, evals, parallelism, prefetching, ...

So context engineering is just one small piece of an emerging thick layer of non-trivial software that coordinates individual LLM calls (and a lot more) into full LLM apps. The term "ChatGPT wrapper" is tired and really, really wrong.

Traduire le post

tobi lutke @tobi · 19 juin
I really like the term "context engineering" over prompt engineering.

It describes the core skill better: the art of providing all the context for the task to be plausibly solvable by the LLM.


Context Engineering
Context
Instructions / System Prompt
Long-Term Memory
State / History (short-term Memory)
Retrieved Information (RAG)
Available Tools
User Prompt
Structured Output

In a way, one could consider that context engineering encompasses all the elements we've studied so far related to AI Agents, from initial instruction to tool definition[12]...

The keys dififculties is that while a prompt is static (at least the template), context is highly dynamic and is generated on the fly depending on the task, some parameters and essentially the state of execution.
The difficulty is to find the right balance between too extremes : providing enough information to be able to carry the task forward correctly without flooding the model with unrelevant information (rising costs and answer fuzzyness).

Context engineering is tantamount to developping the mechanisms that will dynamically fille the model context window before each step of the generation. It can be seen as designing a multiparameter function encompassing all the parameter of the next generation (including the model name and hyperparameters!). This function must be adapted to each specific agent depending on its use case.

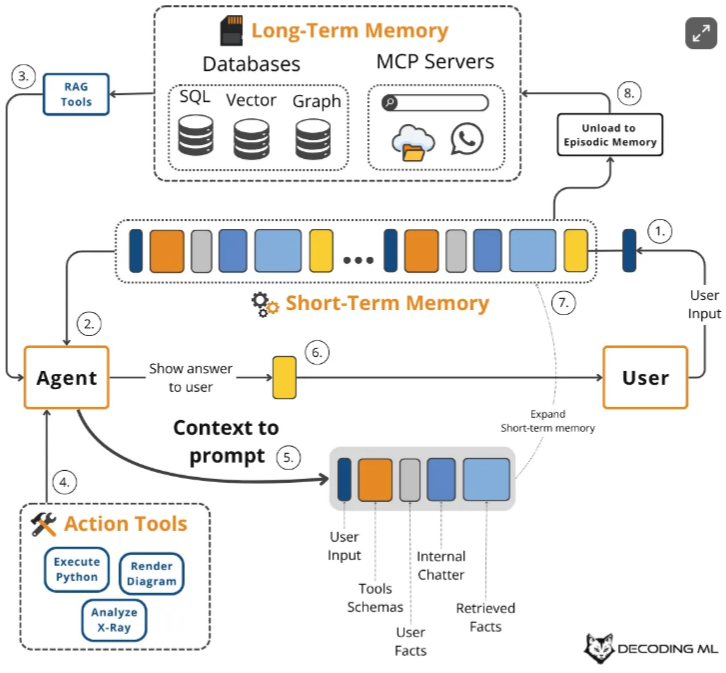
b) Context dynamicity for Agentic AI

After Karpathy's famous post, *everyone* in the community began talking about "context engineering," yet few people attempted to clearly define what the term meant. In this course we choose to treat context engineering as the set of techniques that make the **content of the context window fully dynamic and modular**, rather than a static accumulation of tokens. The idea is that every element inside the window, system prompt, conversation history, memory retrievals, external observations, and examples, can be selected, reformatted, ordered, or discarded according to the agent's needs at each turn.

Several techniques have emerged for this task. At the simplest level, the agent may **maintain a stock of examples** (from past runs or curated demonstrations) and select those most similar to the current situation, much like procedural memory. This selection can be optimized by auxiliary LLMs that reformat or rewrite examples to better fit the new context, or by adjusting the **ordering** of items so that the model pays attention to what matters most. More advanced approaches treat context optimization as an ongoing process: at every turn the agent decides which facts, summaries, or exemplars should enter the window, and how they should be represented. Inference choices—beam search, test-time scaling, reranking—also influence which contextual variants are ultimately used. And underlying it all are the **capabilities of the base model itself**: a model whose training corpus already included many similar examples will make better use of the engineered context than one encountering the task for the first time.

12https://eu.36kr.com/en/p/3366869315372801

Seen this way, context engineering is closely related to **in-context learning (ICL)**. Classic ICL assumed a fixed set of demonstrations provided in a single-turn prompt. In an agentic setting, however, the same principle becomes **multi-turn and dynamic**: the agent continually updates its “few-shot” prompt with the best available exemplars, retrieved facts, or reformatted traces of its own past actions. In other words, ICL becomes one technique among many within context engineering, extended from static prompting into a **dynamic orchestration of memory, examples, and procedures** that evolves with the agent’s run.



Context engineering encompasses many data management strategies[13]. Designing prompts, aka a static template that will be sent to the LLM is not enough anymore, one has to design data stores that will keep track of the agent inner process as well as the state of the external world. From various datastores and tools, a short term memory is built, from which the string sent to the LLM should be derived.

Paradoxically, it is **context engineering** that provides the missing bridge from today’s mostly *stateless* agents to genuinely *stateful* ones. Many popular libraries (e.g. LangChain, Semantic Kernel, even OpenAI’s Swarm) treat agents as essentially stateless processes: each run is independent, context is shallowly passed forward, and “memory” is often reduced to appending conversation history or plugging in a retriever. Some frameworks add a thin layer of observability or persistence, but they lack true flexibility during execution. Early efforts such as **LettaAI** or **Zep** pointed toward a different direction: agents with structured, evolving memories that promise real statefulness. By *real agentic statefulness* we mean the ability to **replay or resume an agent run from any step, modify its components on the fly, and let agents improve automatically by incorporating lessons from previous executions**. This vision reframes agents not as disposable scripts, but as adaptive processes whose context and memories evolve across runs. Once such stateful agents can be composed, shared, and improved, the logic path naturally extends from the **single agent** to the **agentic ecosystem**: a network of agents learning from and building upon one another.

13 https://decodingml.substack.com/p/context-engineering-2025s-1-skill
« It’s a shift in mindset from crafting individual prompts to architecting an AI’s entire information ecosystem »

c) Contextual bottlenecks

While context engineering promises more flexible, goal-aligned, memory-rich agents, there are several unresolved challenges ("bottlenecks") that current research has exposed:

1. **Lost in the Middle / Retrieval Bias**
   Even when very large context windows are supported, LLMs often struggle to recall or pay attention to information buried in the *middle* of the prompt. Research has shown a "U-shaped" performance curve: the beginning and end of the context are more strongly attended, while mid-context content may be underweighted. This leads to forgetting of earlier but still relevant facts.

2. **Context Overload: Noise, Confusion, and Distraction**
   Accumulating too much context (past messages, tool outputs, summaries, etc.) can degrade performance. Irrelevant details become noise, conflicting statements ("context clash") confuse the model, and hallucinations more easily creep in if previous outputs are flawed. The model's ability to prioritize relevant vs irrelevant is still limited. A study (by Databricks) recently showed that contexts stuffed with more than 32k tokens tended to perform significantly worse than shorter ones even if the model context window was wider.

3. **Token Cost & Latency**
   Every additional piece of context costs tokens, which increases prompt size, computational load, memory usage, and inference time. Many research or engineering blog posts note trade-offs but few quantify them comprehensively. Efficient compression, pruning, ranking of context items are required but add complexity.

4. **Dynamicity vs. Modularity Trade-off**
   The promise of being able to select, reorder, edit, or discard context items at each turn adds modularity and flexibility. But this dynamicity requires infrastructure: memory editing tools, retrieval pipelines, context selection policies. Many agents/frameworks offer only static or partially dynamic context, lacking full modularity.

5. **Evaluation Difficulty**
   It's hard to measure whether memory improvements are actually meaningful. Benchmarks for long-term memory, causal coherence, user preference retention are relatively new (e.g. LoCoMo, LongMemEval). Also, different tasks place different demands on memory, so there's no universally accepted metric. Moreover, cost vs accuracy trade-offs are frequently overlooked.

6. **Context Poisoning and Safety**
   When context includes user-provided or retrieved content, flaws like hallucinations, bias, or malicious content can be inserted inadvertently and then influence future outputs. Ensuring that context is not only relevant but *safe* (free of incorrect or inappropriate signals) is a serious challenge. Agents are already being hacked[14]

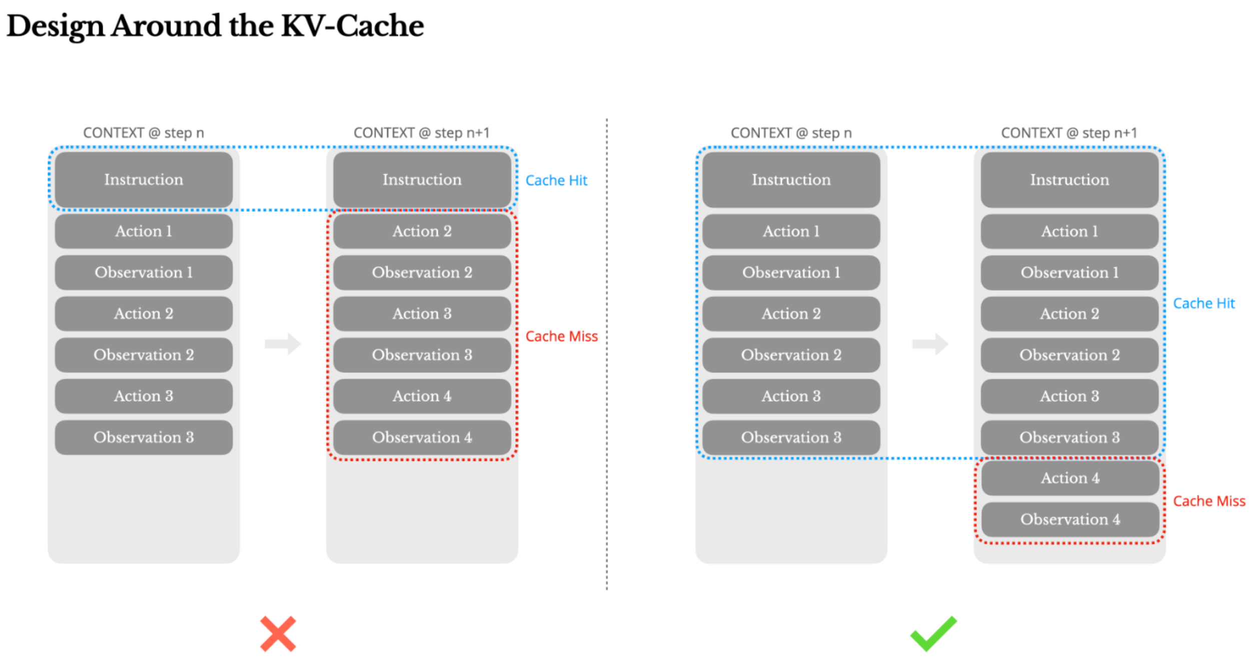
7. **Model Capacity and Training Data Limitations**
   Context engineering assumes the model has been exposed in its training and alignment phases to similar tasks / examples. If not, examples or memory injections may be less

14 https://generativeai.pub/how-a-hacker-tricked-ai-agent-to-send-47-000-0fa7062e1b65

effective. The model's internal capacity (how it encodes patterns from context) limits how well engineered contexts can help. Also, not all models support leaving large context windows or have efficient long-context attention. Centering our agentic architectures around the model themselves make agents fully dependent on model capabilities (and limitations).

The company **Manus** provides a good illustration of how "context engineering" became a buzzword in the agentic space. In July 2024, they published a widely read blog post recounting their experience: after several failed attempts to stabilize their framework, they decided to **bet everything on context engineering**. The story resonated because it captured the reality of many teams: building an agent often comes down to repeated cycles of architectural tweaking, prompt fiddling, and empirical guesswork. Manus humorously dubbed this process *"Stochastic Graduate Descent."*

Their lesson was hard-earned: context engineering is anything but straightforward. They rebuilt their agent framework four times, each time discovering a better way to shape context. One striking empirical finding was the importance of the **KV-cache hit rate**. In a production agent, the input–output ratio is highly skewed: context grows with every step (tool definitions, system prompts, logs, memory), while the output may be just a short function call. At Manus, the average ratio reached **100:1**, meaning that prefill dominates decoding. A small change in the prompt can therefore invalidate the cache from that token onward, wasting resources and slowing execution. Their advice was to treat the context as append-only and to use *breakpoints* deliberately, so that cache reuse is maximized.



Another lesson concerned tools. Manus observed that dynamically adding or removing tool definitions mid-iteration was a bad idea: since most LLMs serialize tool descriptions at the very start of the context (just before or after the system prompt), any change invalidates the KV-cache and breaks continuity. Worse, when previous actions or observations reference tools no longer defined, the model becomes confused. The conclusion: tools are part of the context space itself and must be managed consistently, not shuffled in and out.

Finally, Manus confronted the limits of long contexts. Even with 128k windows, large files or logs could quickly overwhelm the model, degrade performance, and drive up costs. Their solution was to treat the file system as the ultimate context: unlimited in size, persistent by nature, and directly operable by the agent. Instead of forcing everything into the LLM's window, Manus taught the model to read and write from files, using the filesystem itself as structured, externalized memory.

The Manus story shows why context engineering captured so much attention: it is not a matter of abstract principles but of concrete engineering trade-offs : cache management, tool stability, cost of prefilling, and the limits of long contexts. Their "buzz moment" revealed both the promise of context engineering as the key to stateful, scalable agents, and its pitfalls, which remain at the heart of today's contextual bottlenecks.

Conclusion: From Context to Statefulness

The study of agent memory and context engineering reveals a paradox: while LLM agents are often portrayed as intelligent, goal-driven systems, at their core they are still continuous streams of tokens, and optimizing them raises the same problems as evaluating text generation. What matters is not only *what* the model outputs, but *how* successive outputs remain aligned with the agent's overarching goal. This is precisely where memory and context come into play: by shaping what enters the window at each step, they condition every future generation. Evaluating whether this conditioning truly works, however, remains as elusive as evaluating language generation itsel.

The transition from stateless to stateful agents is central here. Stateless frameworks offer little more than long prompts with shallow observability, while stateful agents can learn from their previous runs, replay or modify execution paths, and evolve automatically. In production, this matters enormously: it turns agents from disposable processes into adaptive systems, capable of long-term improvement.

Looking forward, the future of agentic AI lies in richer decision pipelines. Developers will increasingly design agents not by relying blindly on an LLM as the sole "engine," but by integrating deterministic logic (rules, constraints), probabilistic models (Bayesian or decision networks), and neural components. The LLM becomes one action among others , called when its generative flexibility is useful, but bypassed when latency or cost demands efficiency. This reframes context engineering not as a hack to stretch token windows, but as part of a larger discipline: designing modular, stateful, and efficient decision systems.

In this light, context engineering and memory are not just technical tricks: they are the building blocks of agentic ecosystems, where agents carry their goals across turns, learn from experience, and coordinate with one another. The road remains full of bottlenecks (scaling, cost, evaluationbut the direction is clear: from token streams to adaptive agents capable of evolving with their environment.