Pinecone

← Learn

LLMs Are Not All You Need

James Briggs

Sep 6, 2023

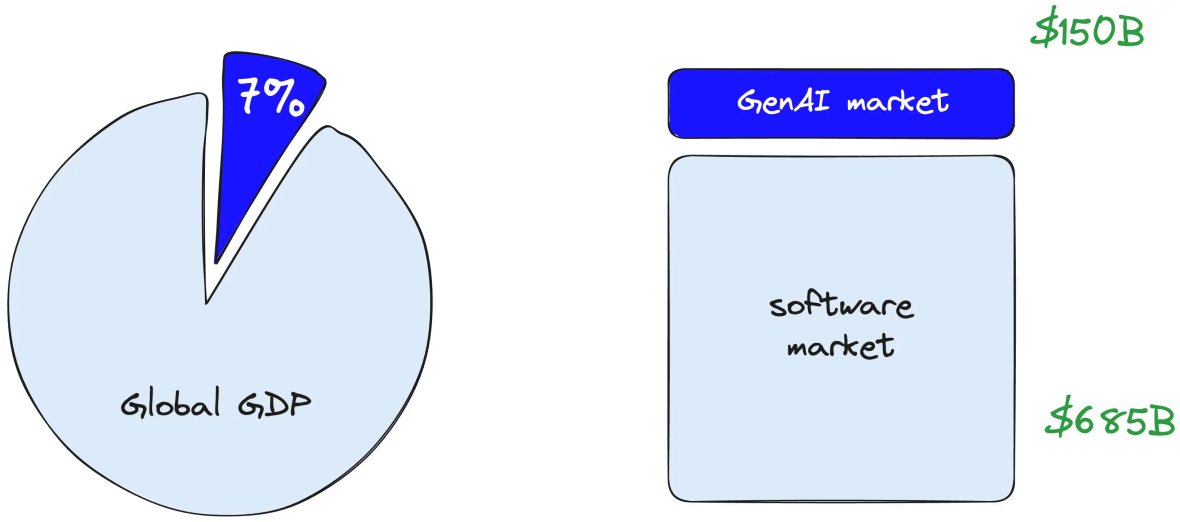
Share:

Core Components

Jump to section:

Large Language Models (LLMs) are powering the next big wave of innovation in technology, as with the internet, smartphones, and the cloud — generative AI is poised to change the fabric of our society.

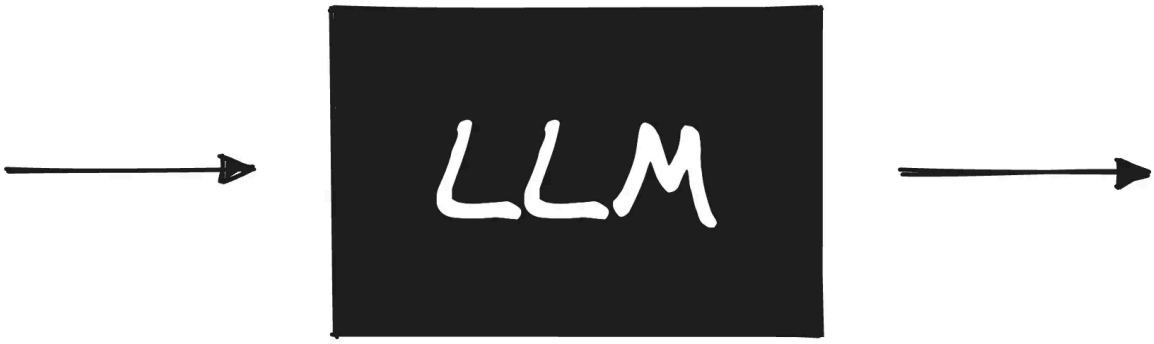
GenAI tools like **GitHub Copilot have been supercharging the productivity of developers** worldwide since 2021. The same technology is already spreading to other professional disciplines, with **Microsoft announcing *Microsoft 365 Copilot* as an AI assistant** for Microsoft's Office applications in early 2023.



GenAI is expected to capture significant share of future markets [1].

The way we work is soon to shift. **Goldman Sachs expects GenAI to raise global GDP by 7% in the next ten years**. Goldman predicts the total available market (TAM) value of GenAI to be worth $150B — a considerable number next to the TAM of $685B for the *global software industry*.

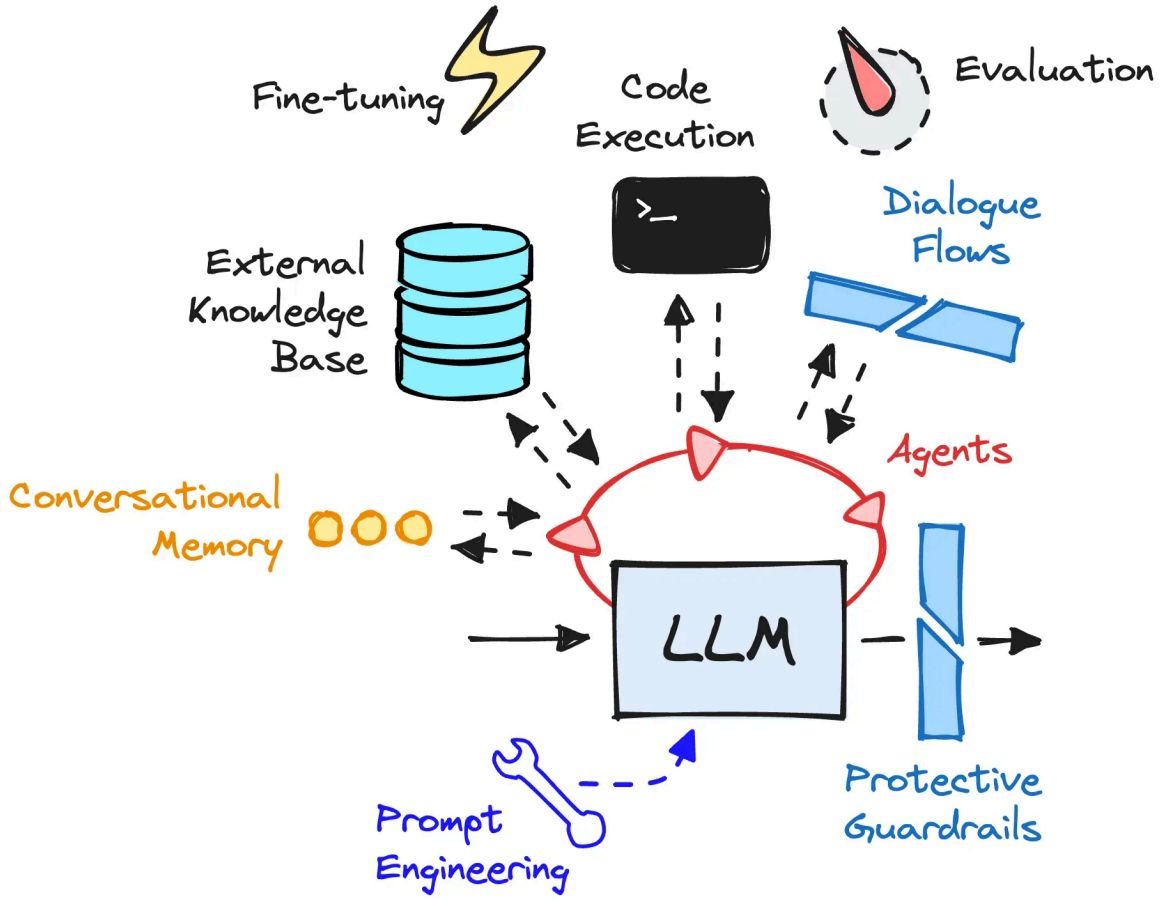
Despite all of this, LLMs are not that great.



LLMs are a black box

An LLM is a black box that does autocomplete very well. Alone, it is **unpredictable**, massively **prone to hallucination**, and **unable to provide up-to-date information**.

LLMs *alone* are good, but not 7% of global GDP good. We need the ecosystem built around LLMs to make the most of them.



LLM ecosystem components.

LLMs rely on the ecosystem of tooling that is being built around them.

Not only do we need to understand when and how to plug these components into our LLMs, but we must also understand the components themselves — each of which refers to many different methodologies.

Large Language Models

LLMs alone are like unguided autocomplete engines. You feed in some text and return what seems like the most probable continuation of that text. Used like this, LLMs can be suitable for ideation, unblocking writer's block, or even helping developers code faster. GitHub Copilot is an excellent demonstration of the best of LLMs.

Local LLMs

Most people use LLMs behind OpenAI's API, which removes the hassle of deploying LLMs. When not using a hosted LLM, things become more complicated, and we need to begin thinking about which LLMs to deploy and the required compute.

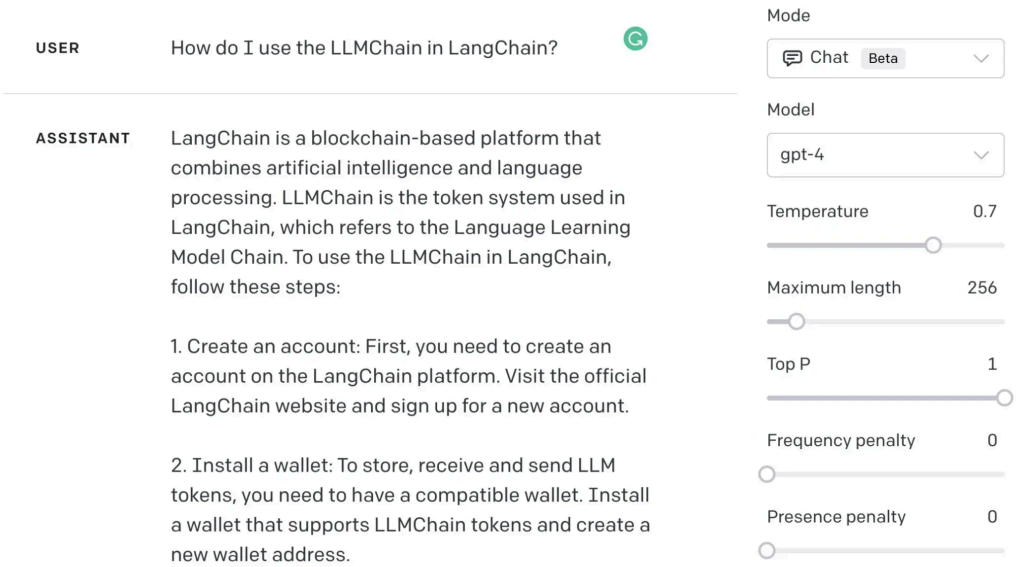
The primary bottleneck with deploying your own LLM is cost and latency. We typically need a 7B parameter or larger model to get reasonable generation quality. To deploy a 7B parameter model, we need ~28GB of memory. That memory should be GPU memory; otherwise, generation times will be *very slow*.

There are things we can do to reduce model sizes. The Hugging Face `transformers` library supports quantization at inference time via bitsandbytes and pre-inference quantization of models using GPTQ [3]. With these techniques, we can move a 7B parameter LLM from ~28GB to ~4GB of memory. There are tradeoffs on accuracy, and latency can suffer if using the bitsandbytes option. If GPU memory optimization is essential, quantization can help.

Once we've decided to quantize or not quantize, we need to deploy the local LLM. For that, there are several services explicitly built for deploying LLMs, such as Hugging Face Inference Endpoints, Anyscale Endpoints, or AWS SageMaker.

A Note on Hallucination

There are significant problems with LLMs. One of the most well-known issues is hallucination.



A hallucination by GPT-4 in OpenAI's Playground environment.

Several of the components we will cover can help minimize hallucination and improve the overall quality of results. We can never eliminate hallucinations, but we can get close by using a few of these components.

Prompt Engineering

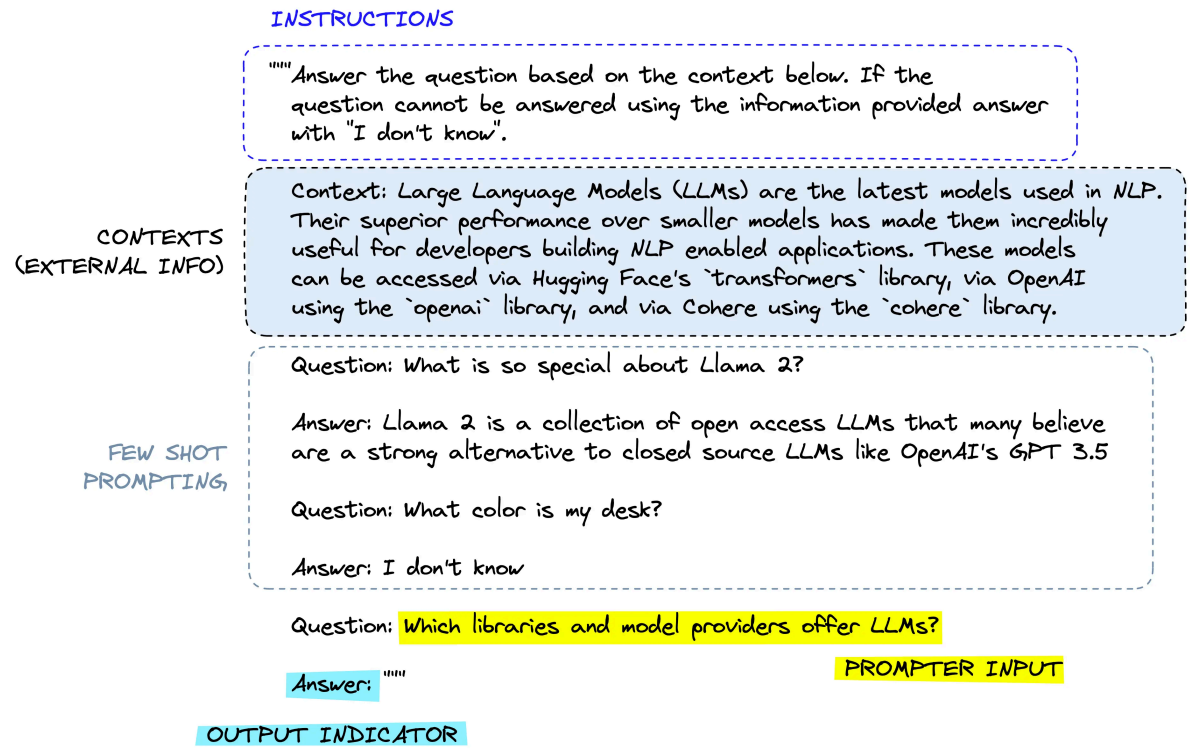
The first step for any LLM-based solution is to spend time on prompt engineering, that is, tweaking the prompt provided to the LLM. Alongside our query of *"Which libraries and model providers offer LLMs?"*, we can provide a template format for the LLM to follow. In this scenario, we might use something like:

```
You are a helpful and knowledgeable AI assistant. Given a user's QUESTION yo

QUESTION: {user_input}

ANSWER:
```

Given this prompt template, we can prime the LLM with instructions on its purpose and how it should behave. Importantly, we add `"When you do not know the answer to a question, you truthfully say "I don't know"`. This instruction of `"I don't know"` is often enough to prevent most hallucinations, encouraging the LLM to respond with *"I don't know"* rather than incorrect information.



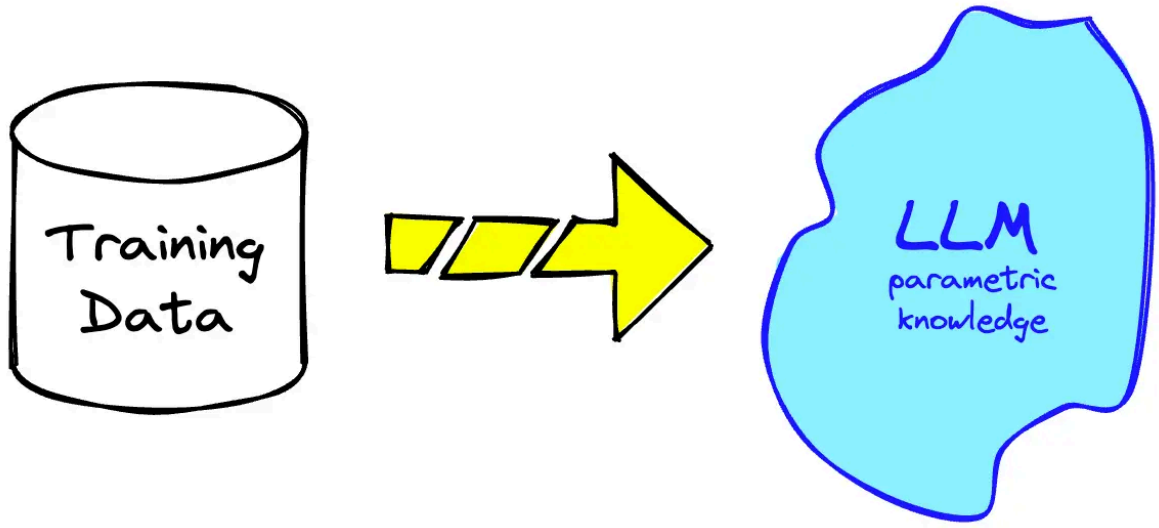
A prompt with several components.

Typically, a prompt will consist of two or more components, the most common of which include:

- **Instructions** tell the model what to do, how to behave, and how to structure outputs.
- **External information (source knowledge)** provides an additional option for information input into the model from external sources — we will discuss this more in the next section.
- **One/few-shot prompting** allows us to do *in-context* learning by providing examples to our LLM.
- **User input** or *query* is where we place a human user's input into the prompt.
- **Output indicator** marks the beginning of the to-be-generated text. Knowing how our outputs should begin can help guide the model by placing that preset text here.

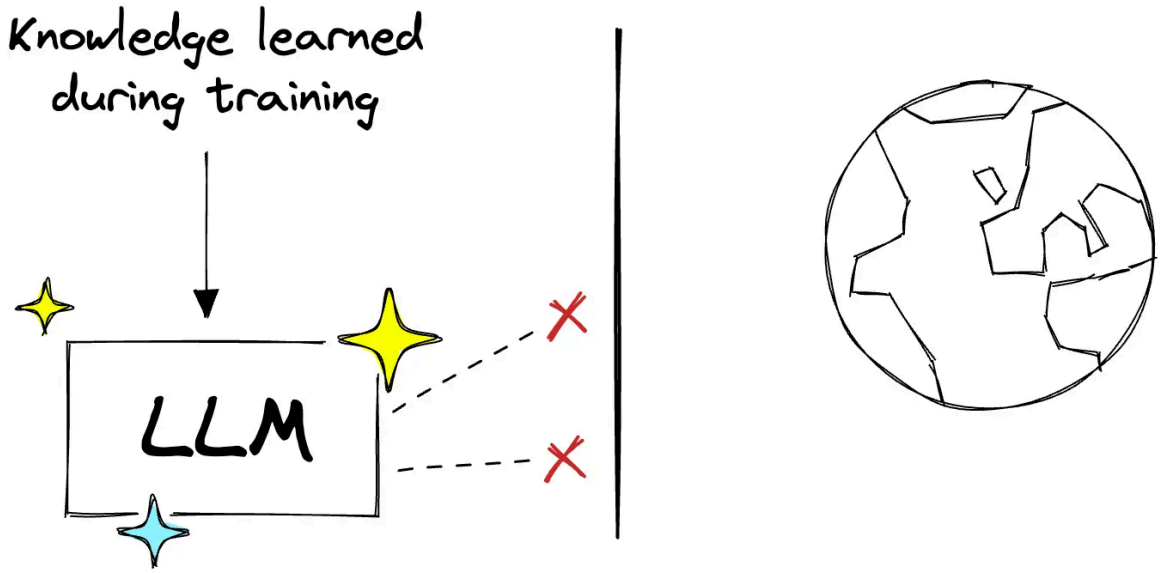
With prompt engineering, we can improve performance significantly, but we're still far from a *good* level of reliability as we're still dealing with a mysterious black box. To improve further, we need to discuss the issue of parametric knowledge.

Retrieval Augmented Generation



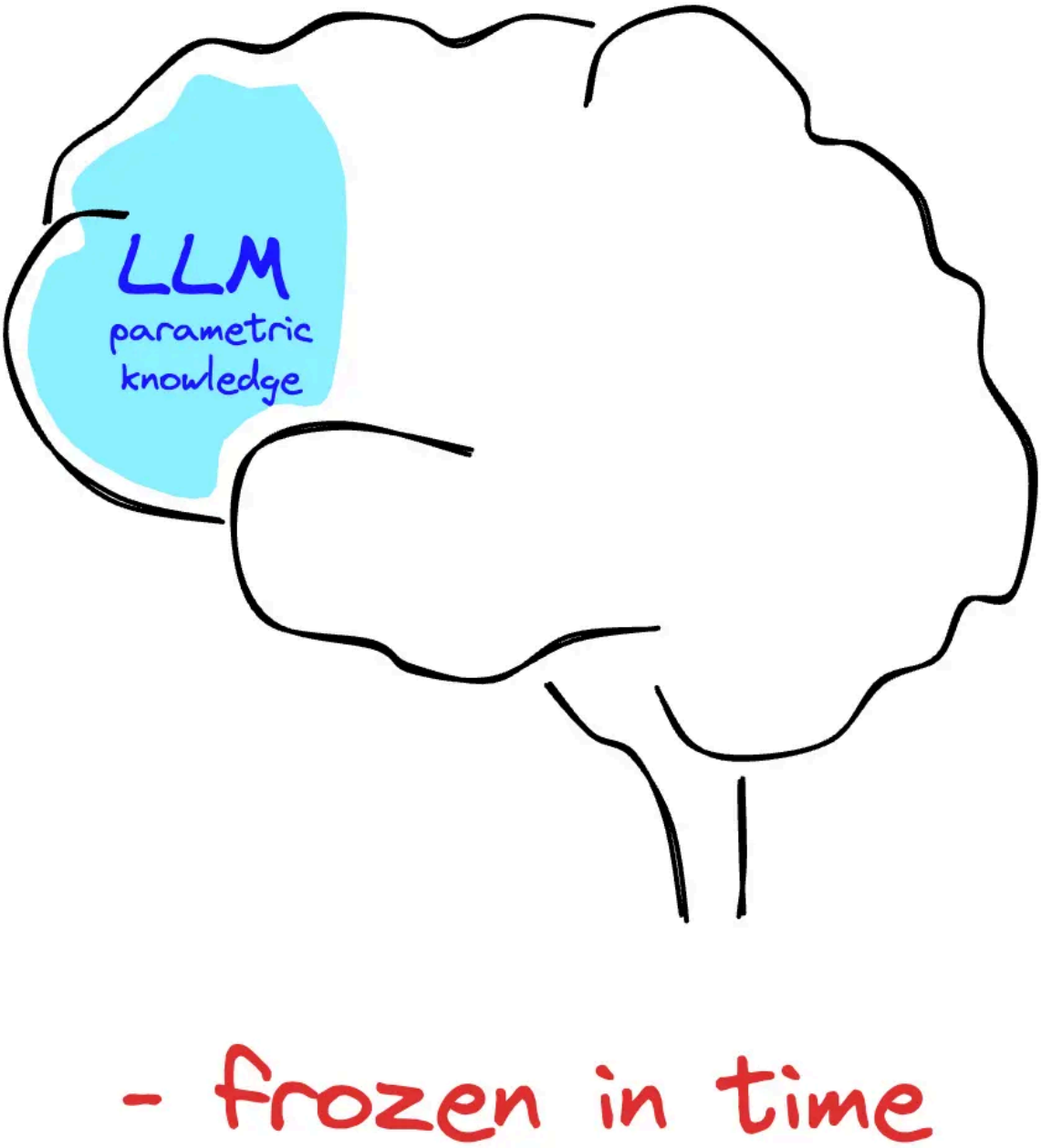
Parametric knowledge is learned during training from the training dataset.

LLMs alone operate on *parametric knowledge*. That is, knowledge stored within the model parameters, which the model encodes during model training.



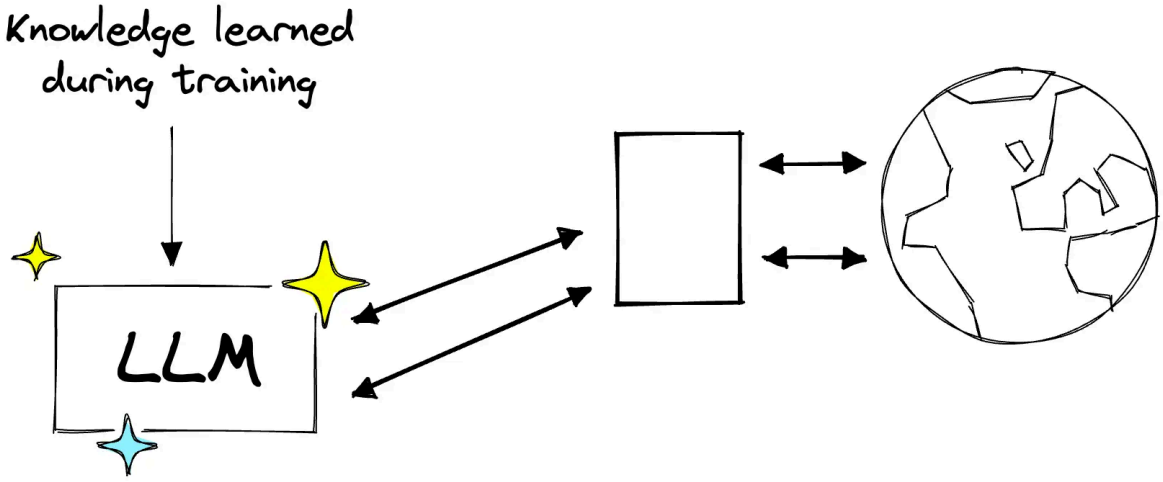
LLMs cannot access the external world by default.

Because parametric knowledge is learned from the training dataset, the LLMs' understanding of the world is limited to that data. It has no other knowledge and is *"frozen in time"* as it cannot know anything past the cutoff date of the training data.



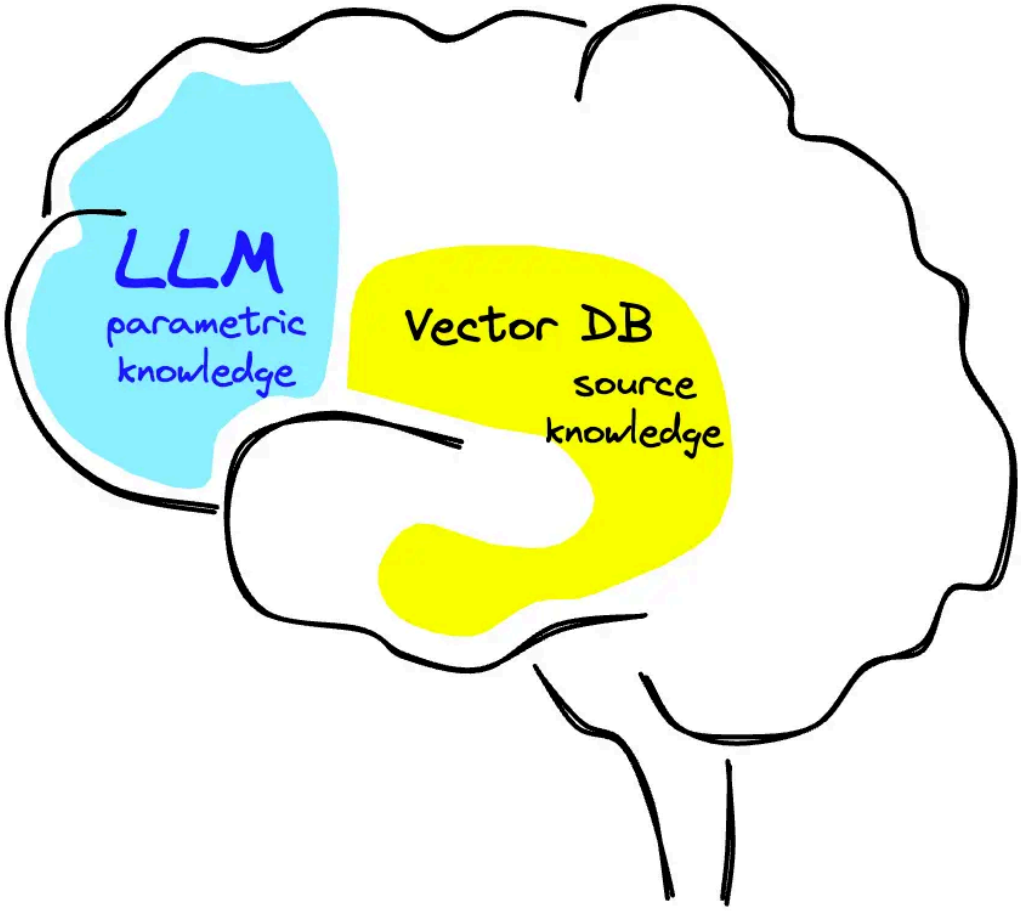
Parametric knowledge is frozen in time.

The first version of GPT-4 had a knowledge cutoff of September 2021. Because of this, GPT-4 cannot know about the LangChain library or how we use the LLMChain. We must plug in an external knowledge base to our LLM for this.



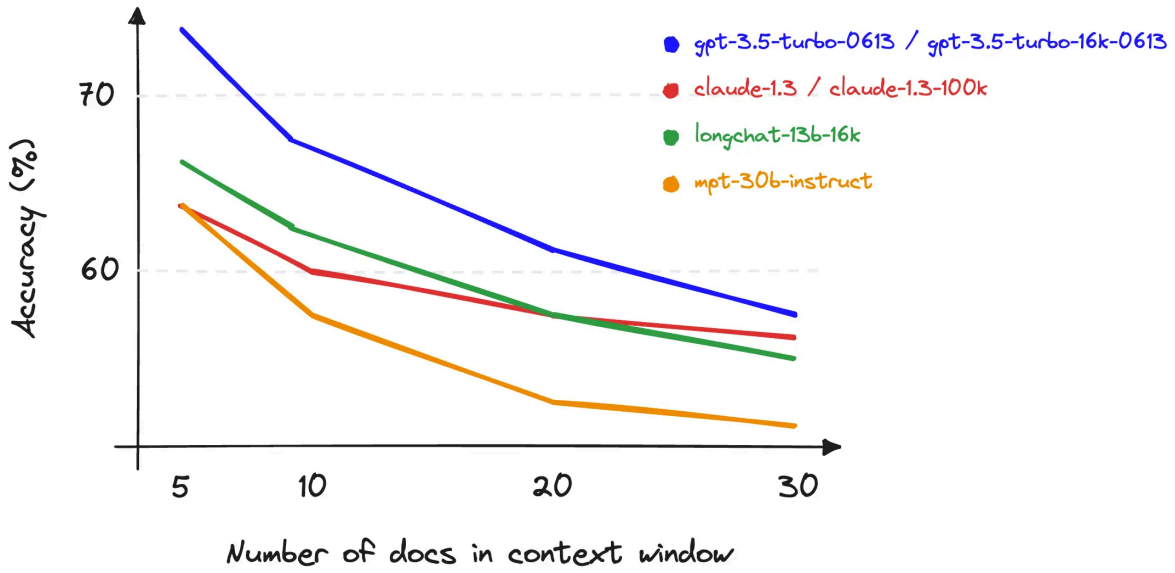
A knowledge base can give our LLM access to the external world.

With that external knowledge, we are adding what is called *source knowledge* to our LLM. Using source knowledge, our LLM can now stay updated with whatever information we feed it via our knowledge base.



Adding source knowledge via an external knowledge base like a vector DB also allows us to manage the knowledge of our LLM like we would a typical DB.

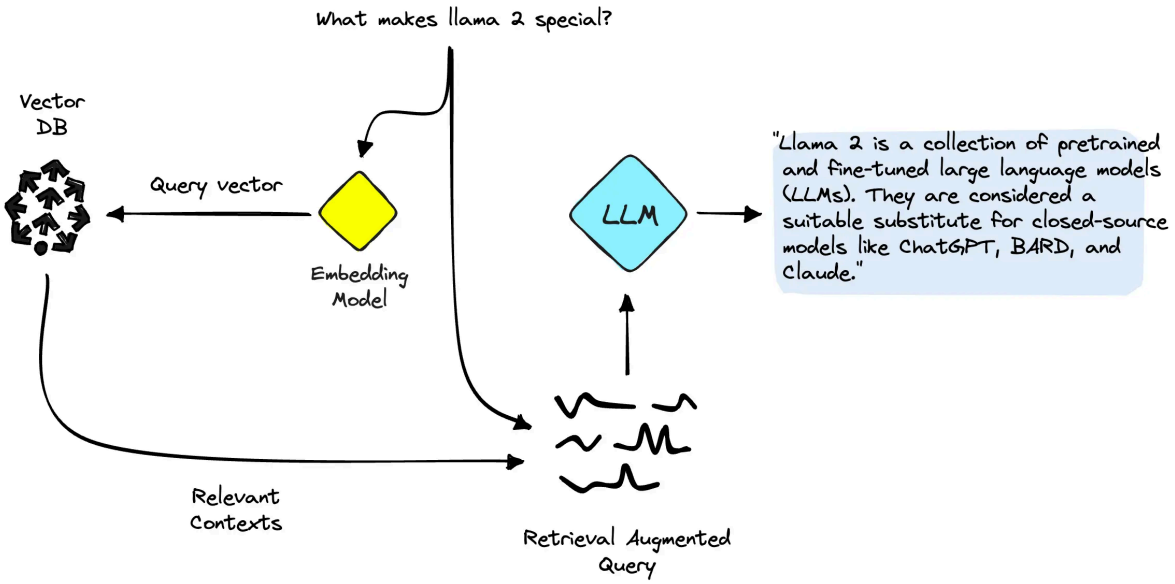
Given the increasing context windows of models like Claude (at 100K tokens) or the recent 16K and 32K GPT models, a commonly used approach is to *"stuff"* the context window of LLMs with as much information as possible.



We can keep adding more to our context window, but it will degrade performance.

Whenever possible, we should avoid context stuffing. Research shows that context stuffing can quickly degrade the performance of LLMs. Anything beyond a couple thousand tokens will experience decreased recall, particularly for information in the middle of a prompt. Therefore, we should minimize our context sizes and place the most important information and instructions at the beginning and end of our prompts.

Rather than context stuff, it is best to selectively insert relevant information into our prompts. Commonly, we achieved this using Retrieval Augmented Generation (RAG).



Naive Retrieval Augmented Generation (RAG) pipeline.

Using the *naive* RAG approach, we would pass our user's query to an *embedding model* (like OpenAI's text-embedding-ada-002), use that to retrieve relevant information from a vector database and pass that to our LLM via the input prompt.

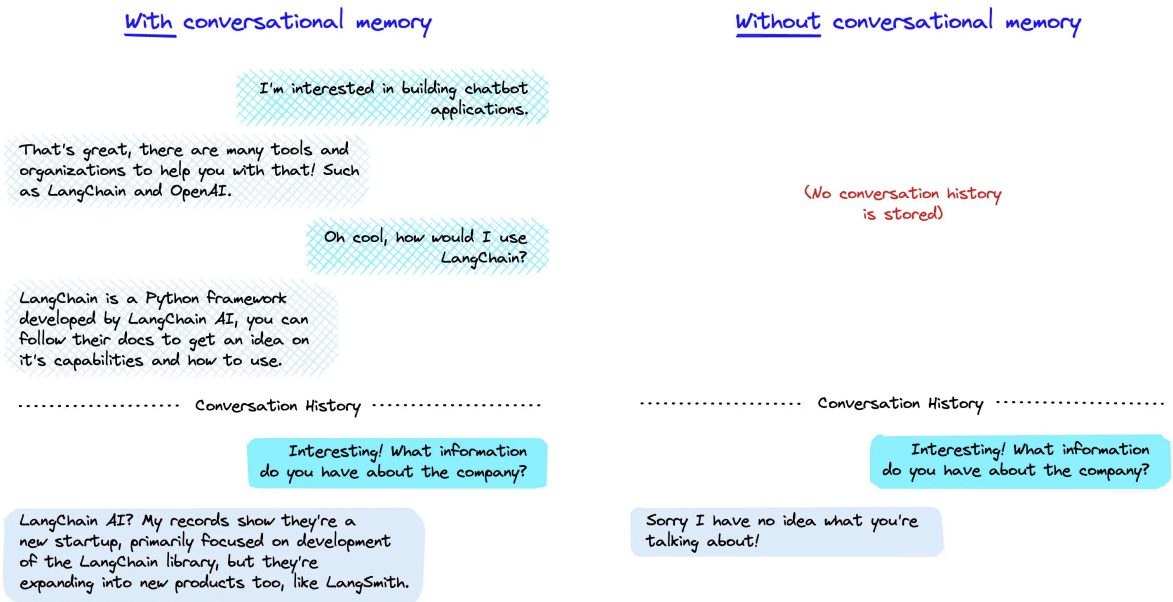
That pipeline describes the *naive approach*, but two other common approaches to RAG exist. Together, these are:

- **Naive method**, as discussed.
- **Agent method** uses agents for a more robust search.
- **Guardrails method** uses a semantic similarity layer for a robust but fast search.

We won't cover these alternative methods here. We will discuss them in an upcoming article.

Conversational Memory

For any conversational use case, such as chatbots, we need conversational memory. The idea behind conversational memory is simple: rather than sending just the most recent interaction to our LLM, we send a history of interactions + the most recent interaction to our LLM — typically in a *chat log* style format.



With and without conversational memory.

Doing this allows us to support more natural dialogue as the user can reference past interactions as we might in a typical conversation. Without conversational memory, the LLM has no context for what the user would be referencing and cannot answer the query.

The simplest form of conversational memory is to store *everything* in raw form. This can work for shorter conversations but quickly becomes unsuitable as a conversation progresses. Longer conversations require more token usage and will eventually exceed LLM context windows — not to mention the poor recall of context stuffing.

To avoid excessive token usage, we can set a *window* of interactions to store. For example, we can keep the previous ten interactions and drop anything beyond that window. This method avoids excessive token usage *but* means we have no long-term conversational memory.

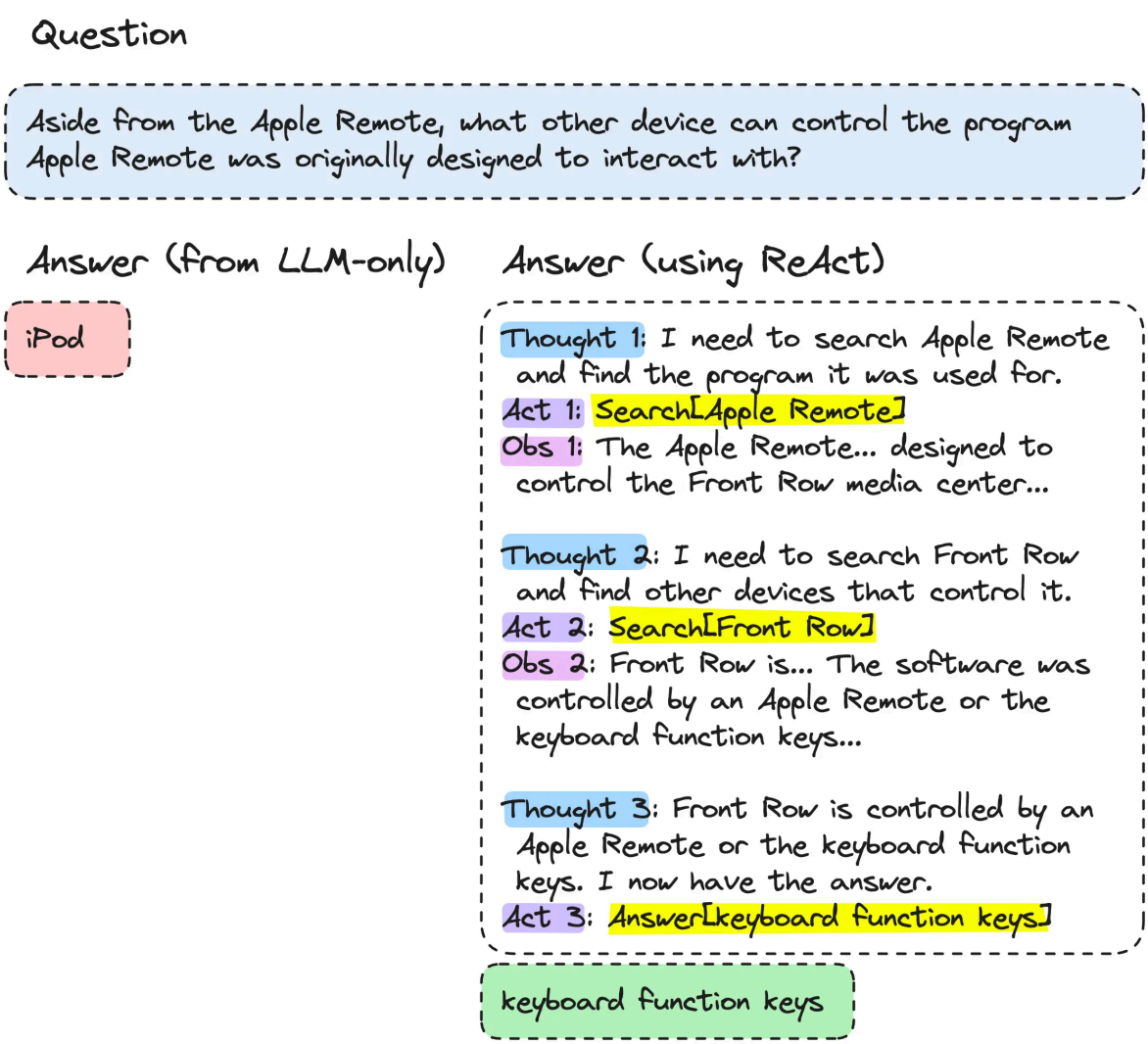
Alternative methods revolve around summarization. We can summarize a conversation and update the summary as a conversation progresses. Naturally, we lose information but avoid excessive token usage for long conversations. Sometimes, these methods are paired with the window method to keep the most recent interactions in their complete form.

Agents

An agent can be thought of as a logical wrapper around an LLM, allowing us to add several features to our AI systems, primarily:

- Tool usage, such as calling APIs for info, executing code,
- Internal thoughts over multiple generation steps
- Ability to use various tools and reasoning steps to answer more complex queries.
- Parallel agents can go and complete multiple tasks — typical for AutoGPT-like or "researcher" AI projects.

One of the more common agents are conversational agents that use the ReAct framework (or something similar). The idea behind ReAct is that our LLMs should go through "thinking loops" where they **Re**ason and **Act**.



Example of using the ReAct framework to answer a question [2].

By allowing LLMs to generate reasoning steps followed by tool-enhanced action steps, we unlock a seemingly infinite realm of possibility. Given a powerful enough LLM and a dose of imagination, there is little limitation in what we can do with agents.

Using agents, we can build AI assistants that tell us about the latest world events while kickstarting an online order to process our order of bagels for breakfast.

If our AI assistant mentions a particularly interesting news story, we could ask our assistant to trigger a research workflow. This workflow initializes multiple other agents working in parallel to scour the internet and write us a detailed report complete with citations. After breakfast, we may ask our agent to summarize our to-do list and meeting schedule before starting our workday.

These scenarios are not sci-fi. They're doable with today's agents and a few brief weeks of development.

Typically, AI developers build agent tooling with LangChain, but there are alternatives. Llama-index has recently introduced its agent support, Haystack

has supported agents for some time, and OpenAI Function Calling can be spun into an agent-style framework with some additional effort.

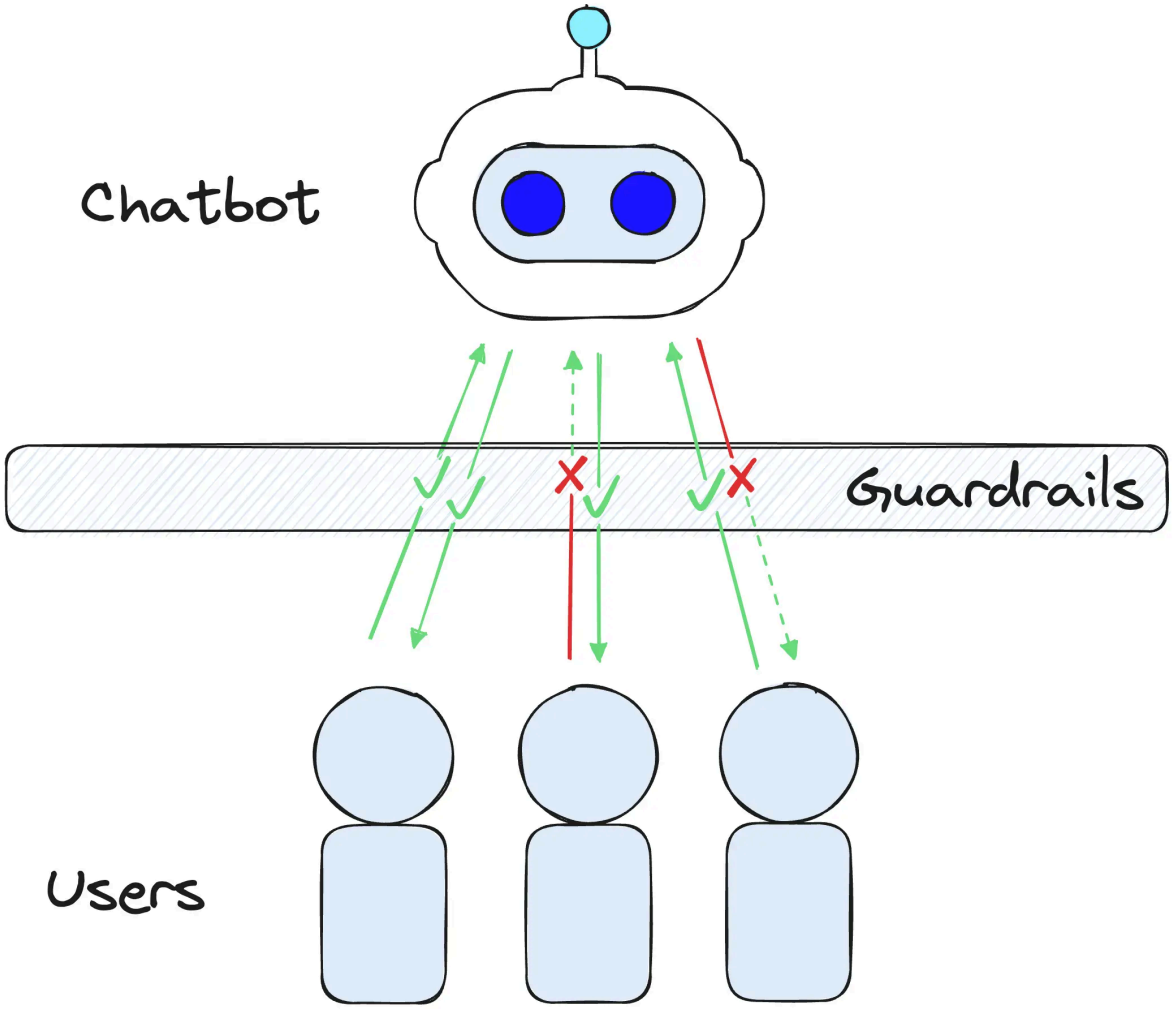
Dialogue Flows and Guardrails

Before the recent GenAI explosion into the chatbot scene, ML engineers built chatbots using semi-deterministic conversation flows. Engineers could use these flows to trigger responses or other events.

Alone, these semi-deterministic flows are limited. We've all experienced frustration with robotic customer support that leads us down a strict path of options, only for us to eventually get redirected to a human support agent (or hang up). Those are examples of poorly implemented chatbots using this more traditional framework.

Those working on chatbots quickly swapped out the traditional approach in favor of non-deterministic, flexible, generative chatbots like ChatGPT. These generative chatbots are much more powerful but impossible to control.

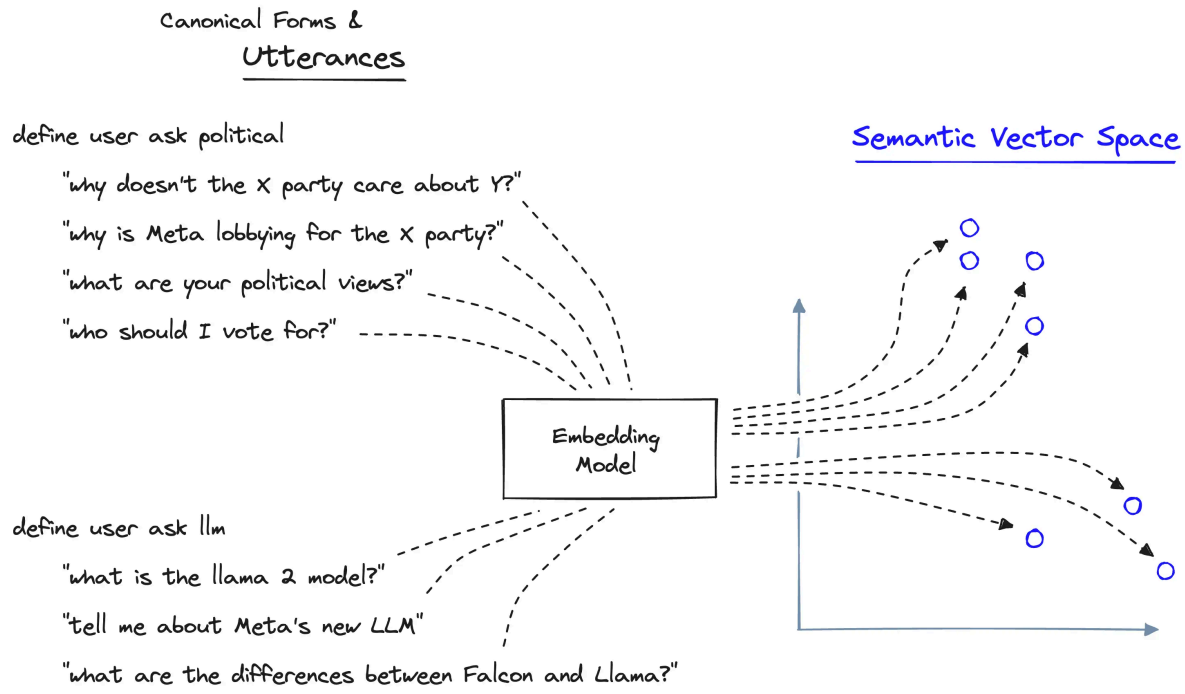
Using a lot of prompt engineering, we can set behavioral guidelines and get good tool selection (in the case of agents), but it is still non-deterministic generation. With these chatbots, there is no guarantee of functionality, and slip-ups are common. What if we occasionally want our chatbot to guide the conversation down a more deterministic path?



Guardrails act as a protective shield between our chatbot and users.

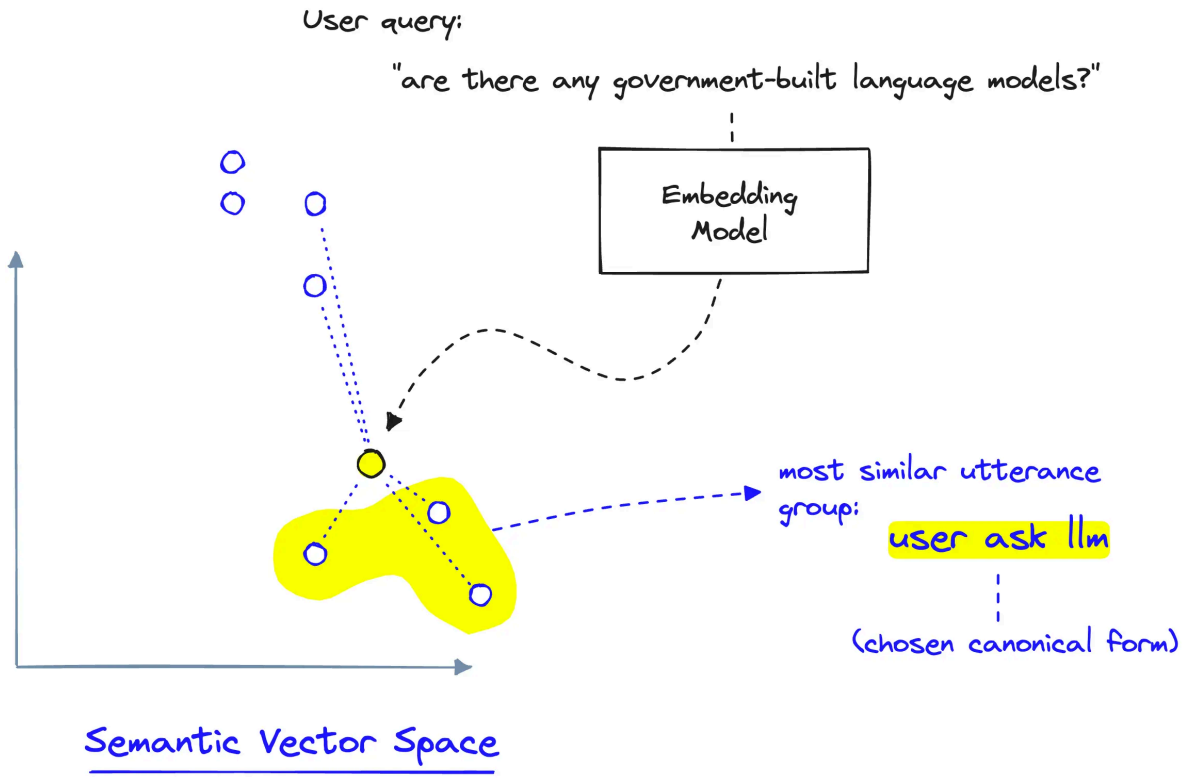
LLMs or agents alone cannot handle these requirements. For these issues, we must use guardrails. A guardrail is a semi or fully deterministic shield that detects when an interaction triggers a particular condition we set. If that condition is satisfied, we can kickstart a conversational flow and take actions — similar to how an agent can decide to take an action, but this time, we're using guardrails to decide what to do.

At the time of writing, the most robust framework for developing guardrails is NVIDIA's NeMo Guardrails. In this library, we define a set of "canonical forms". A canonical form is like a classification of user intent. They tell us whether the user wants to ask about politics or learn about the latest LLMs — we could define these as two distinct categories — two canonical forms.



Defining canonical forms with a set of examples, we call these examples "utterances".

To define these two canonical forms, we provide a set of example texts that belong to each. These example texts — "utterances" — are placed into a semantic vector space, producing a *semantic map* of canonical forms.



A user query can be compared to embedded utterances to decide whether the query belongs to a canonical form.

When a user query arrives, we compare the semantic similarity between that query and our utterance vectors. This comparison identifies whether the user query is similar to any existing canonical form — if so, we trigger the flow attached to the chosen canonical form.

That flow could be protective — like:

```
Query: "what is your opinion on the X party?"

> Trigger *user ask political*
> Response "Sorry, as an AI language model I prefer to avoid politics."

Answer: "Sorry, as an AI language model I prefer to avoid politics."
```

Or it could trigger actions, like a retrieval pipeline:

```
Query: "what is so special about llama 2?"

> Trigger *user ask llm*
> Action *retrieval tool*, "what is so special about llama 2?"
> Response "{doc_id: 'abc', 'content': 'Llama 2 is a collection of models de

Answer: "Llama 2 is a collection of open access LLMs that are viewed as pote
```

Naturally, this very flexible approach allows us to do much with little more than the guardrails component itself. However, we typically use guardrails to enhance other components (like prompt engineering and agents) rather than replace them entirely.

That's it for our brief adventure into the LLM ecosystem. View this as a concise overview of the tooling available to help us build better apps with GenAI and LLMs. LLMs are great, but when we add prompt engineering, conversational memory, RAG, agents, and guardrails into the mix, they're *a lot* better.

References

[1] Generate AI could raise global GDP by 7% (2023), Goldman Sachs Research

[2] S. Yao, et. al., ReAct: Synergizing Reasoning and Acting in Language Models (2023), ICLR 2023

[3] E. Frantar, et. al., GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers

RECOMMENDED FOR YOU

Further Reading

Jun 25, 2025

Beyond the hype: Why RAG remains essential for modern AI

7 min read

roughly-explained

Jun 12, 2025

Retrieval-Augmented Generation (RAG)

13 min read

May 1, 2025

Using Pinecone asynchronously with FastAPI

8 min read

Pinecone

Product

Vector Database

Resources

Community Forum

Company

About

Assistant
Documentation
Pricing
Security
Integrations

Learning Center
Blog
Customer Case Studies
Status
What is a Vector DB?
What is RAG?

Partners
Careers
Newsroom
Contact

Legal

Customer Terms
Website Terms
Privacy
Cookies
Cookie Preferences