2025 EDITION

FREE

AI Engineering

System Design Patterns for LLMs, RAG and Agents

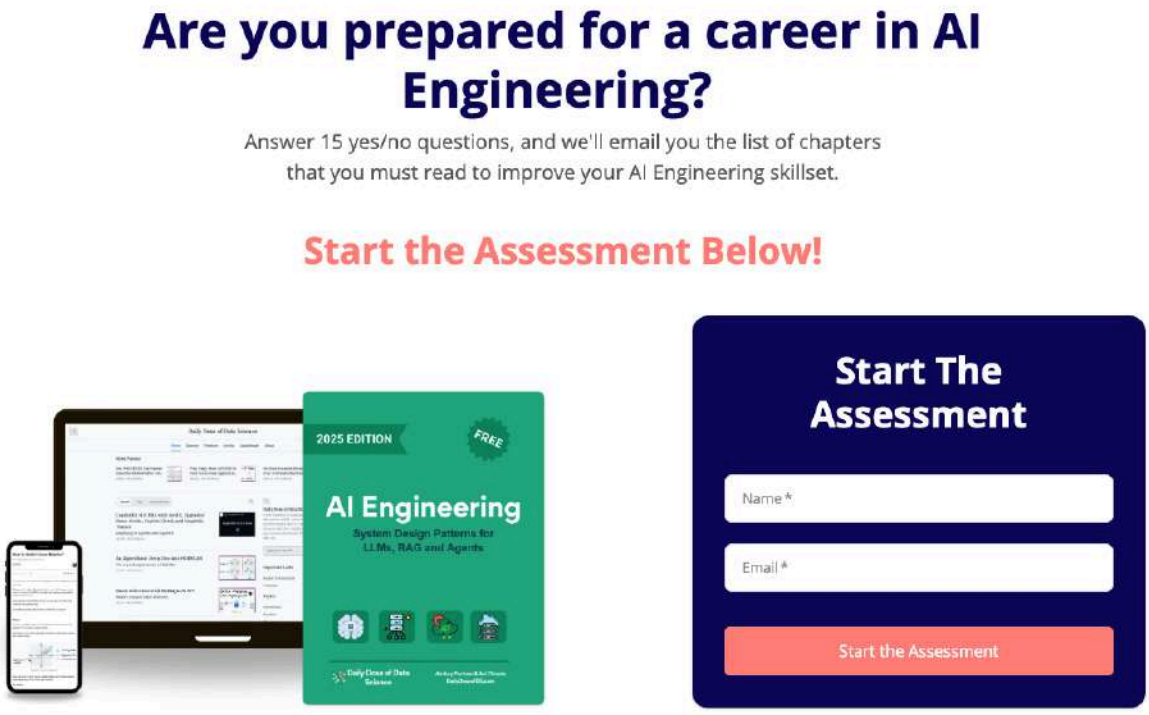
Daily Dose of Data Science

Akshay Pachaar & Avi Chawla

DailyDoseofDS.com

How to make the most out of this book and your time?

The reading time of this book is about 20 hours. But not all chapters will be of relevance to you. This 2-minute assessment will test your current expertise and recommend chapters that will be most useful to you.



Scan the QR code below or open this link to start the assessment. It will only take 2 minutes to complete.

https://bit.ly/ai-engg-assessment

AI Engineering

Table of contents

LLMs

What is an LLM?

Imagine someone begins a sentence:

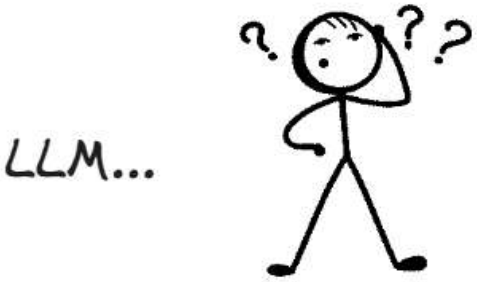
"Once upon a..."

You naturally think "time."

Or they say:

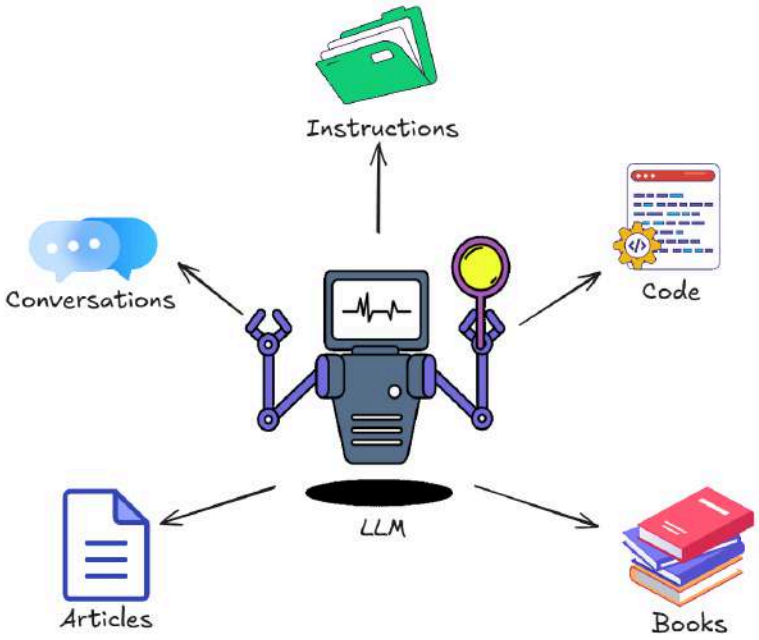
"The capital of France is..."

You immediately think "Paris."



This simple act of predicting what comes next is the foundation of how large language models(LLMs) operate.

They learn to make these predictions by reading enormous amounts of text: books, articles, scientific papers, code, conversations, and instructions.

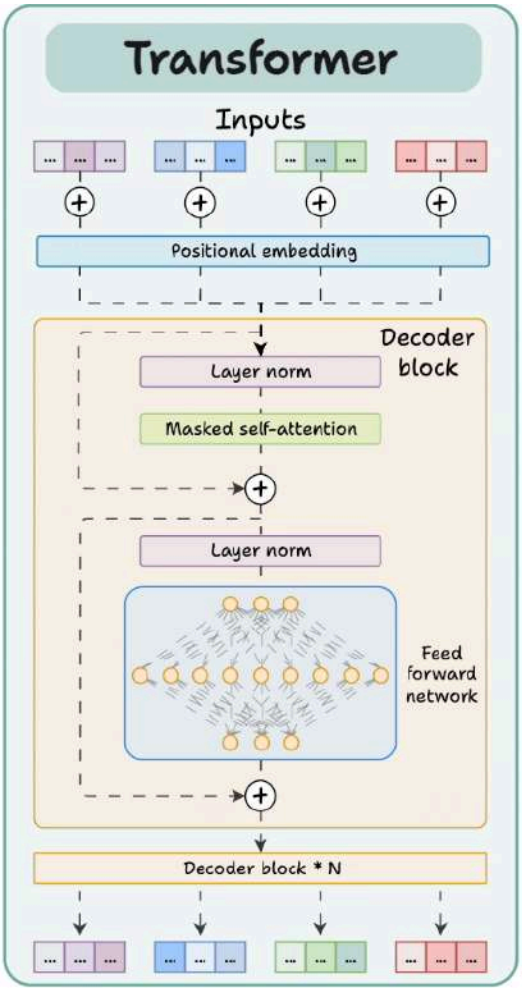


With enough exposure, the model becomes remarkably good at continuing any piece of text in a coherent, meaningful way.

At the technical level, an LLM processes text in small units called tokens. A token may be a word, part of a word or even punctuation.

The model looks at the tokens so far and predicts the next one. Repeating this process generates full answers, explanations, or code.

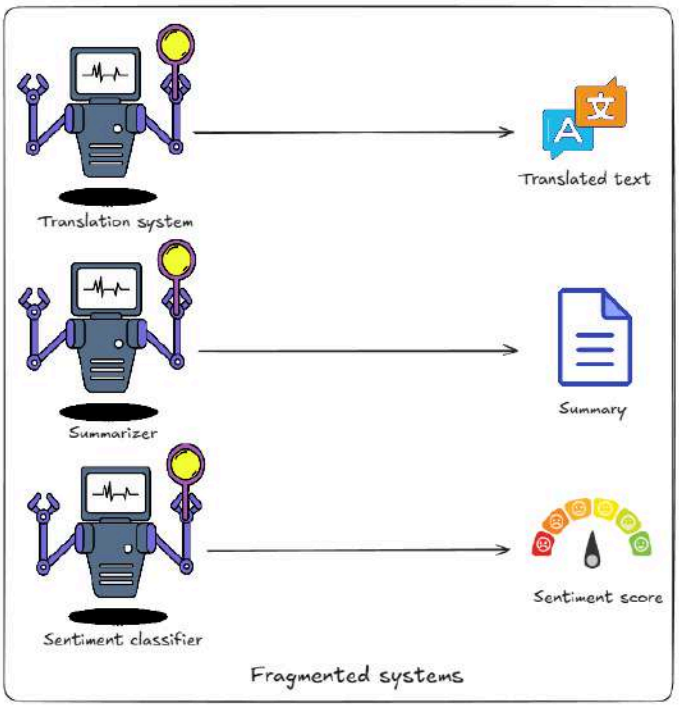
Everything an LLM does from summarizing a document, generating a function or explaining a concept emerges from choosing the next token that best fits the patterns it has learned.



To formalise, a large language model is a Transformer-based neural network trained on massive text corpora to predict the next token in a sequence and through this process acquires the ability to understand, generate and reason with human language.

Need for LLMs

Before LLMs, AI systems were built for specific tasks.

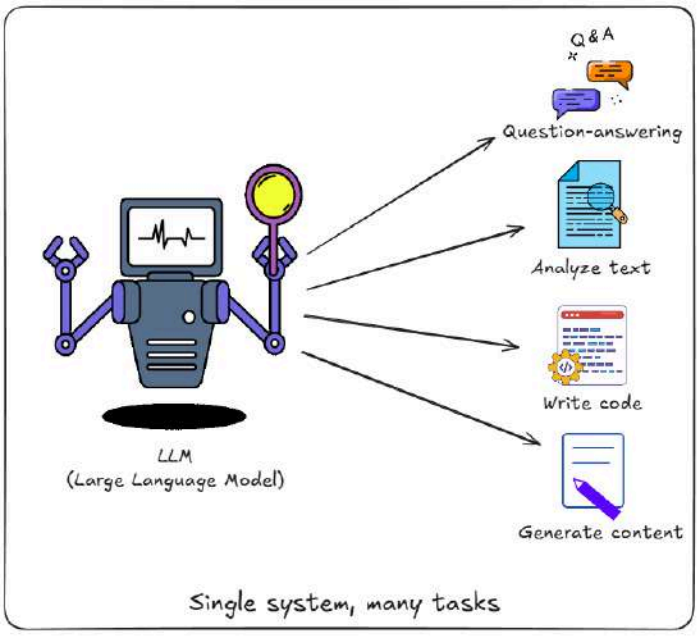


- A translation system handled only translation.
- A summarizer knew only summarization.
- A sentiment classifier recognized only sentiment.

Each new problem required a new model and a new pipeline. This created a fragmented landscape that didn't scale.

LLMs changed this.

By learning the general structure of language across millions of domains, one model could suddenly perform many tasks without being explicitly programmed for each one.

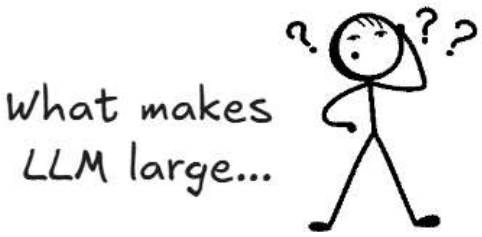


Language naturally encodes reasoning steps, factual knowledge, explanations and communication patterns. Training a model on large enough text collections allowed it to internalize these patterns and apply them across tasks.

As a result, a single system could now answer questions, write code, analyze text and more simply by predicting the continuation of your input.

What makes an LLM 'large' ?

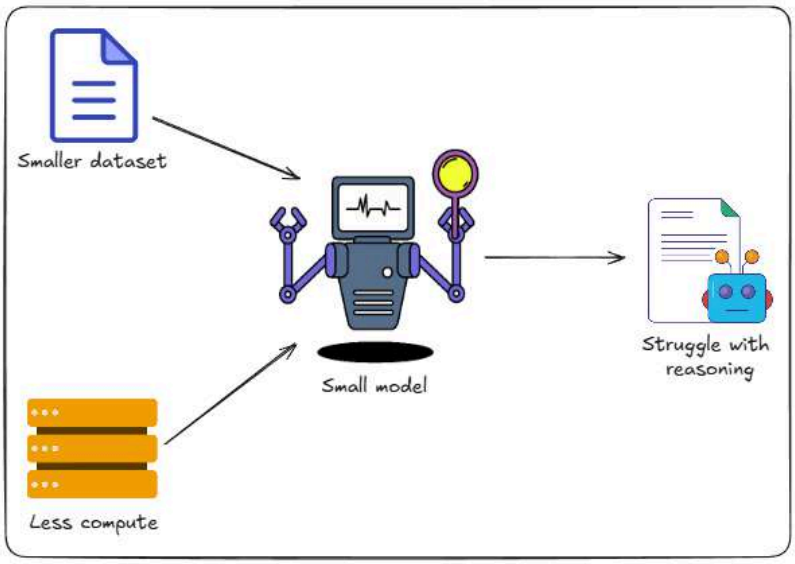
When we call a language model "large," we are referring to its scale:



- Number of parameters it contains
- Amount of data it has been trained on
- Compute used to train it

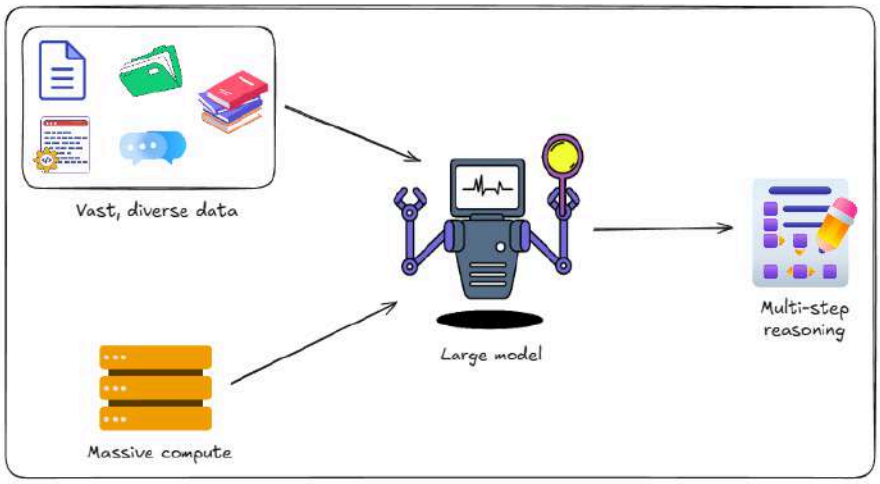
Parameters are the internal values that the model adjusts during training. Each parameter represents a small piece of the patterns the model has learned.

Earlier language models were much smaller and could only capture surface-level text patterns.



They could mimic style but struggled with tasks that required reasoning, abstraction or generalization.

As researchers increased model size, dataset diversity and training compute, a clear shift appeared.



Larger models began to follow detailed instructions, perform multi-step reasoning and solve problems they had never encountered directly in training.

This wasn't the result of adding new rules or programming specific behaviors. It emerged naturally from giving the model enough capacity to learn deeper relationships in language.

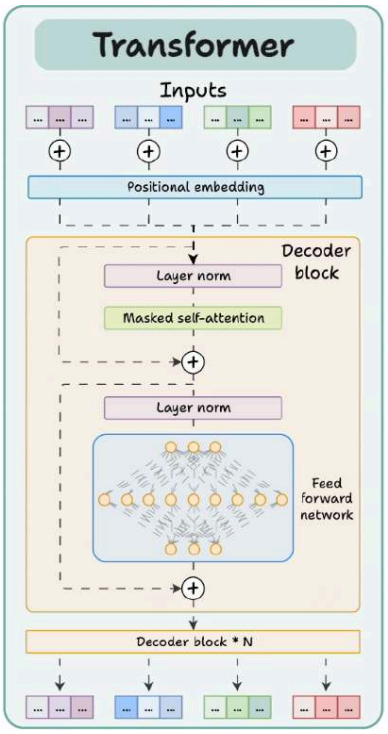
This effect held consistently: models with more parameters, trained on broader data, produced more reliable, coherent and adaptable outputs.

In practical terms, the "large" in large language model is what enables these capabilities.

How are LLMs built?

Before an LLM can be trained, it needs an architecture that can process text, learn patterns and scale across large datasets.

This architecture is built from several core components that work together to turn raw text into structured representations the model can learn from.



Transformer

At the center of modern LLMs is the Transformer.

A Transformer is designed to look at all tokens in the input at once and identify which parts of the text are most relevant to each other.

This lets the model follow long sentences, track references, and understand relationships that appear far apart in the sequence.

Tokenization

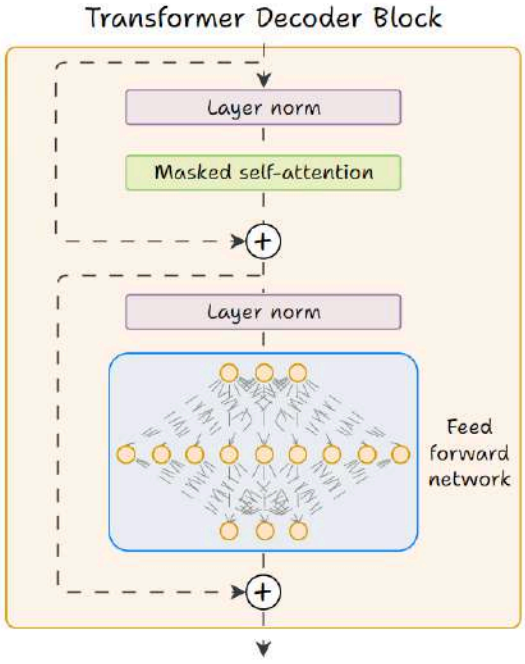
Text is first broken into tokens. A token may be a word or part of a word, depending on how common it is.

This approach keeps the vocabulary manageable and allows the model to handle any language input.

These tokens are then mapped to numerical representations so the model can work with them.

Transformer Layers

The model contains many Transformer layers stacked on top of each other.

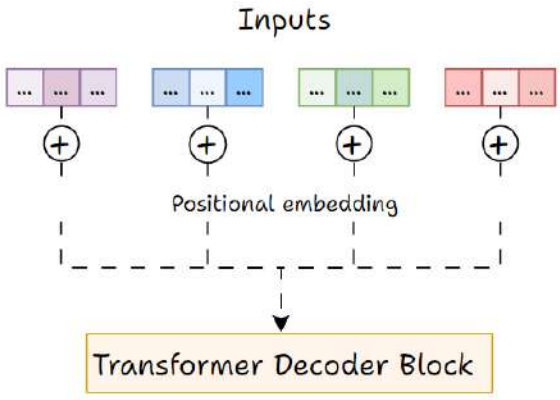


Each layer refines the model's understanding by comparing tokens, attending to important parts of the input, and updating their representations.

As the sequence moves through these layers, the model builds a deeper view of the text.

Positional Encoding

Transformers do not naturally know the order in which tokens appear.



To provide this information, positional encodings are added to the token representations.

These encodings give the model a sense of sequence, enabling it to interpret ordered structures such as sentences, lists, or code.

Parameters

Inside the architecture are parameters - the values the model adjusts during training.

A large LLM contains billions of these parameters.

They store the patterns the model learns from text and form the basis for its ability to understand and generate language.

Distributed Training Setup

Because these models are too large for a single machine, they are trained across many GPUs in parallel.

The model's parameters, computations and training data are distributed so the system can process massive datasets and update billions of parameters efficiently.

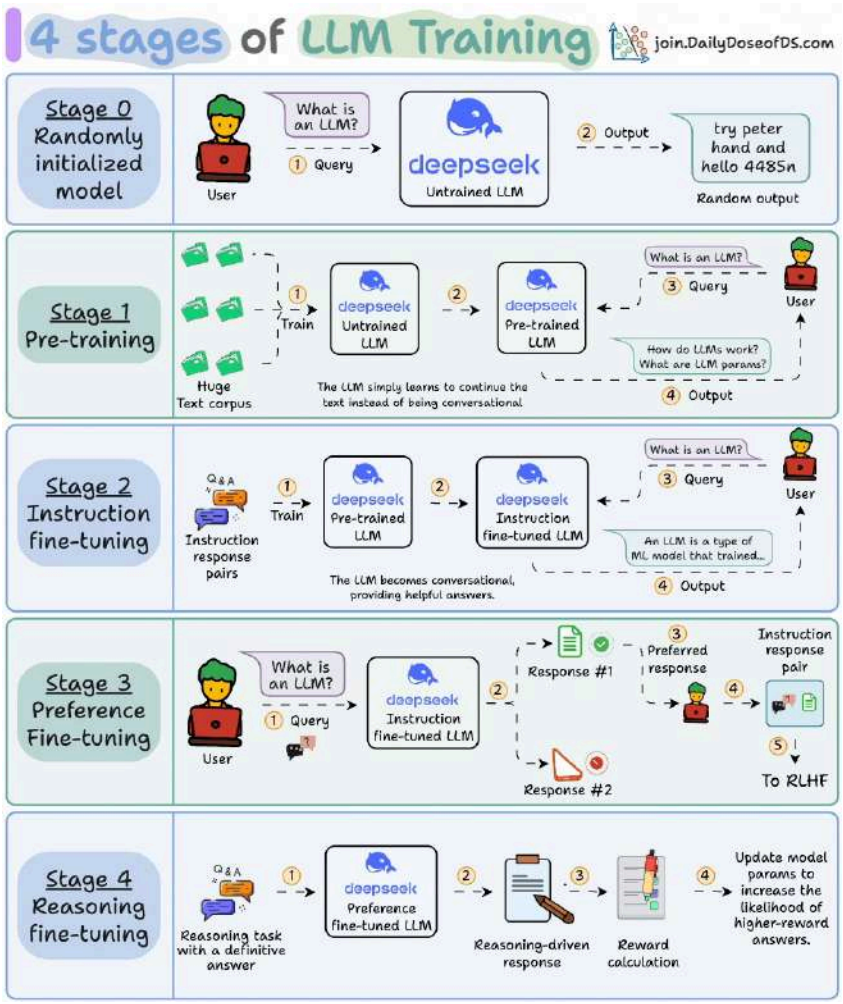
How to train LLM from scratch?

After the model is built, the next step is to train it so it can understand language and handle tasks that appear in real-world use cases.

We'll cover:

- Pre-training
- Instruction fine-tuning
- Preference fine-tuning
- Reasoning fine-tuning

The visual provides a clear summary of how these stages fit together.



0) Randomly initialized LLM

At this point, the model knows nothing.

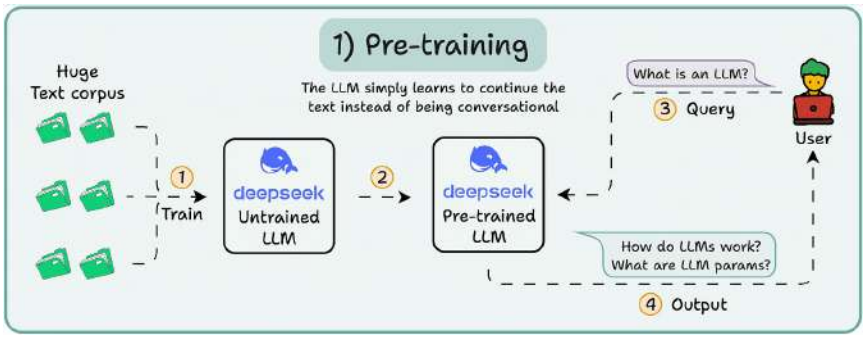
You ask, "What is an LLM?" and get gibberish like "try peter hand and hello 448Sn".

It hasn't seen any data yet and possesses just random weights.

1) Pre-training

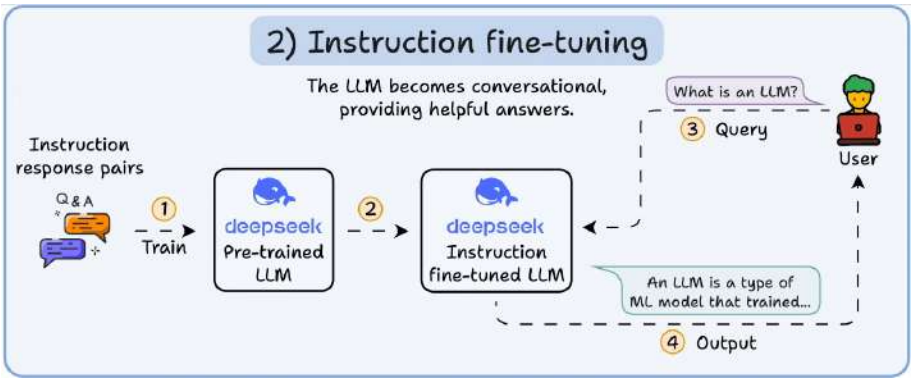
This stage teaches the LLM the basics of language by training it on massive corpora to predict the next token. This way, it absorbs grammar, world facts, etc.

But it's not good at conversation because when prompted, it just continues the text.



2) Instruction fine-tuning

To make it conversational, we do Instruction Fine-tuning by training on instruction-response pairs. This helps it learn how to follow prompts and format replies.



Now it can:

- Answer questions
- Summarize content
- Write code, etc.

At this point, we have likely:

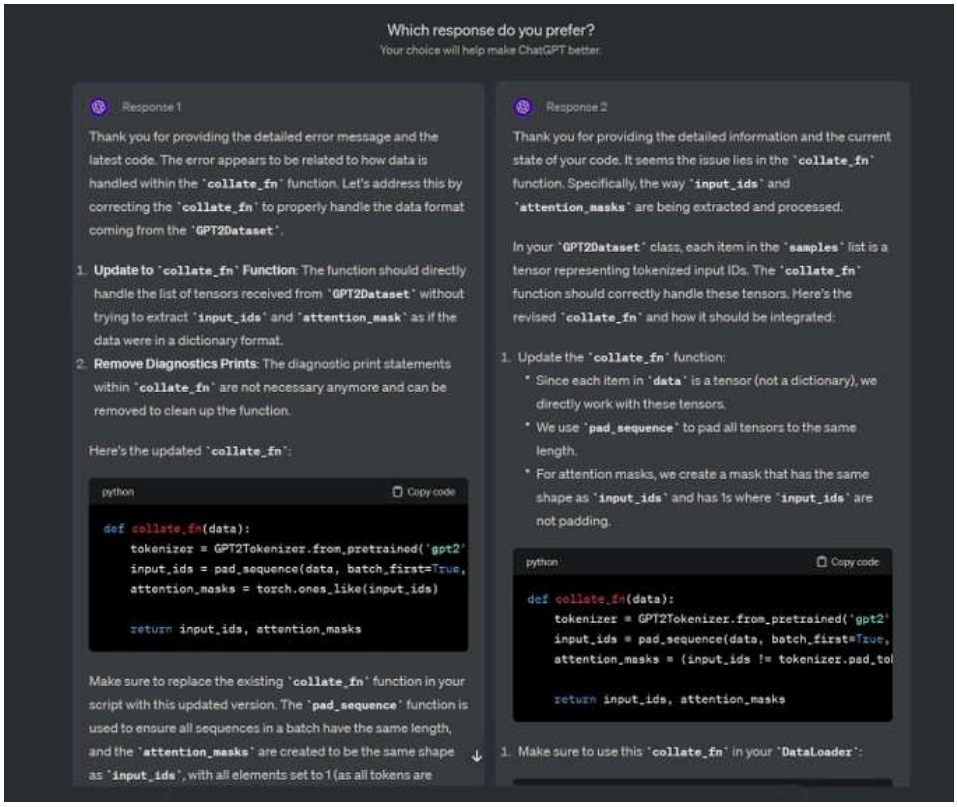
- Utilized the entire raw internet archive and knowledge.
- The budget for human-labeled instruction response data.

So what can we do to further improve the model?

We enter into the territory of Reinforcement Learning (RL).

3) Preference fine-tuning (PFT)

You must have seen this screen on ChatGPT where it asks: Which response do you prefer?



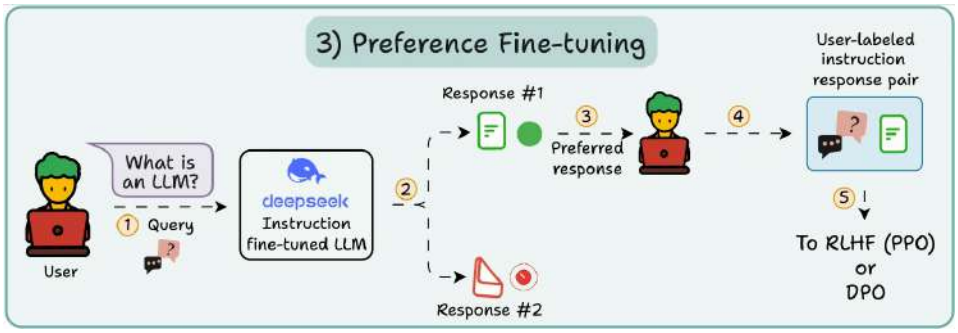
That's not just for feedback, but it's valuable human preference data.

OpenAI uses this to fine-tune their models using preference fine-tuning.

In PFT:

The user chooses between 2 responses to produce human preference data.

A reward model is then trained to predict human preference and the LLM is updated using RL.



The above process is called RLHF (Reinforcement Learning with Human Feedback), and the algorithm used to update model weights is called PPO.

It teaches the LLM to align with humans even when there's no "correct" answer.

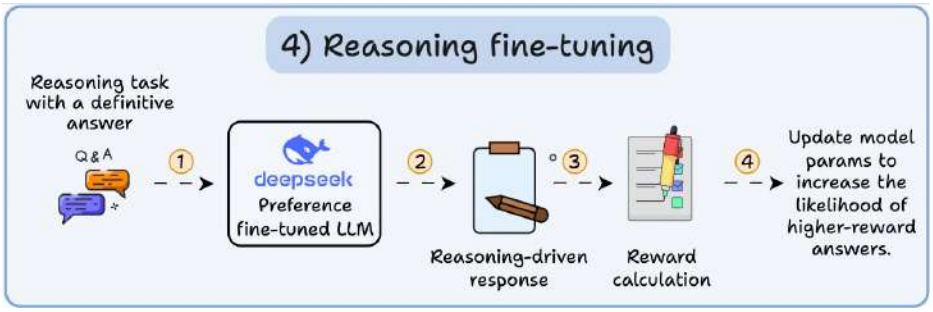
But we can improve the LLM even more.

4) Reasoning fine-tuning

In reasoning tasks (maths, logic, etc.), there's usually just one correct response and a defined series of steps to obtain the answer.

So we don't need human preferences, and we can use correctness as the signal.

This is called reasoning fine-tuning



Steps:

- The model generates an answer to a prompt.
- The answer is compared to the known correct answer.
- Based on the correctness, we assign a reward.

This is called Reinforcement Learning with Verifiable Rewards. GRPO by DeepSeek is a popular technique for this.

Those were the 4 stages of training an LLM.

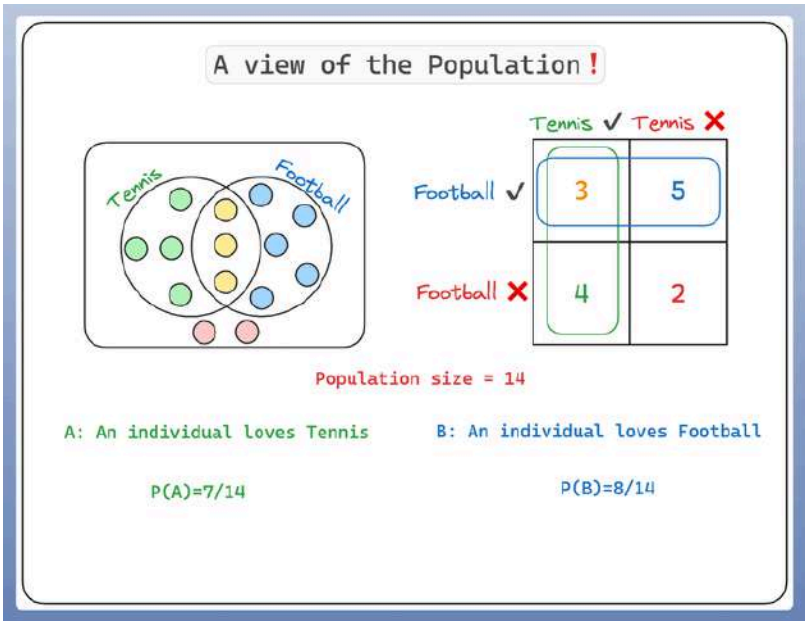
- Start with a randomly initialized model.
- Pre-train it on large-scale corpora.
- Use instruction fine-tuning to make it follow commands.
- Use preference & reasoning fine-tuning to sharpen responses.

How do LLMs work?

Let's understand how exactly LLMs work and generate text.

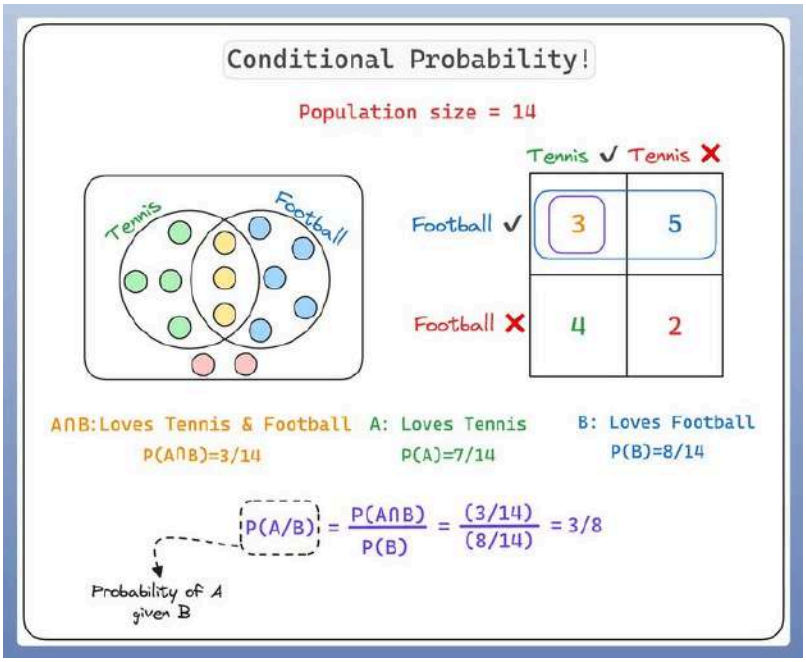
Before diving into LLMs, we must understand conditional probability.

Let's consider a population of 14 individuals:



- Some of them like Tennis
- Some like Football
- A few like both
- And few like none

Conditional probability is a measure of the probability of an event given that another event has occurred.



If the events are A and B, we denote this as P(A|B).

This reads as "probability of A given B"

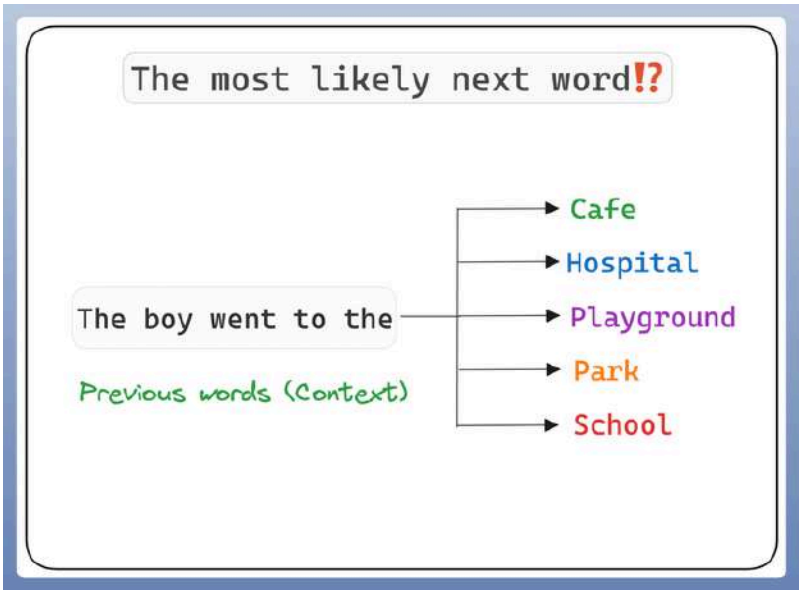
For instance, if we're predicting whether it will rain today (event A), knowing that it's cloudy (event B) might impact our prediction.

As it's more likely to rain when it's cloudy, we'd say the conditional probability P(A|B) is high.

That's conditional probability!

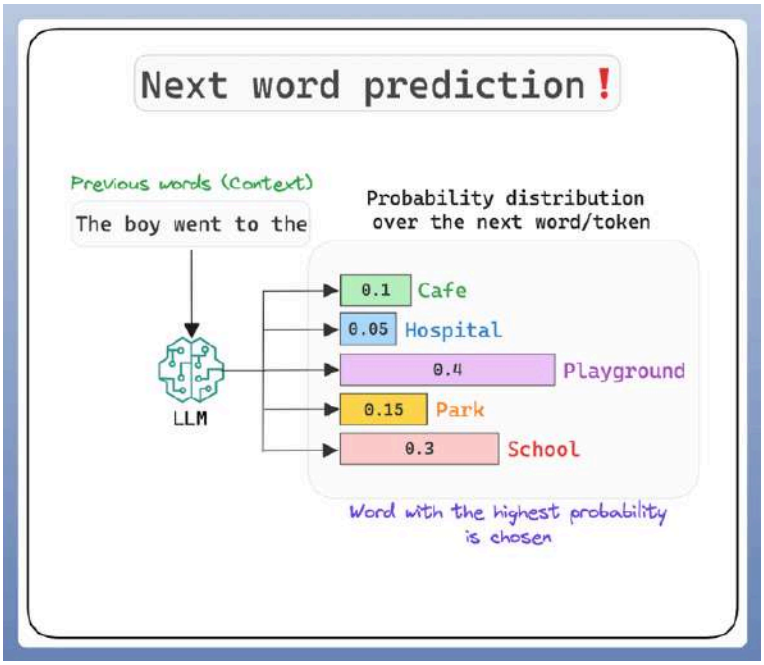
Now, how does this apply to LLMs like GPT-4?

These models are tasked with predicting/guessing the next word in a sequence.



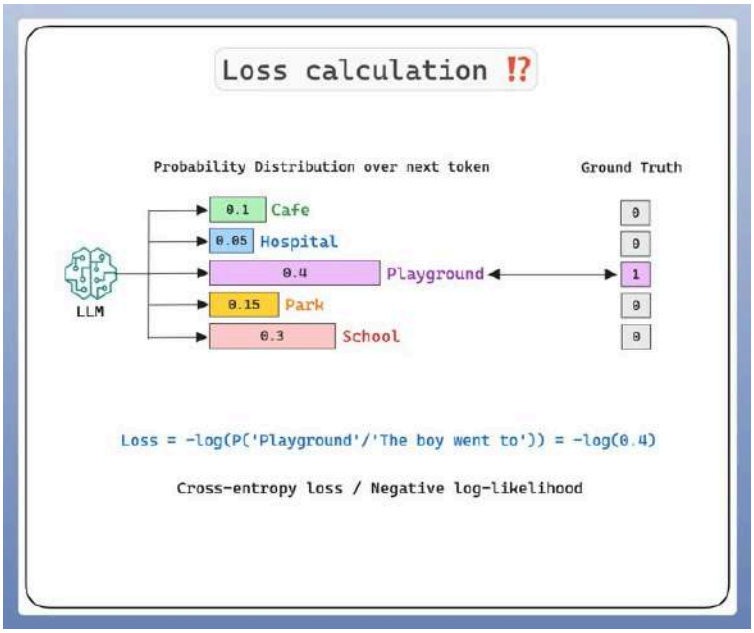
This is a question of conditional probability: given the words that have come before, what is the most likely next word?

To predict the next word, the model calculates the conditional probability for each possible next word, given the previous words (context).



The word with the highest conditional probability is chosen as the prediction.

The LLM learns a high-dimensional probability distribution over sequences of words.

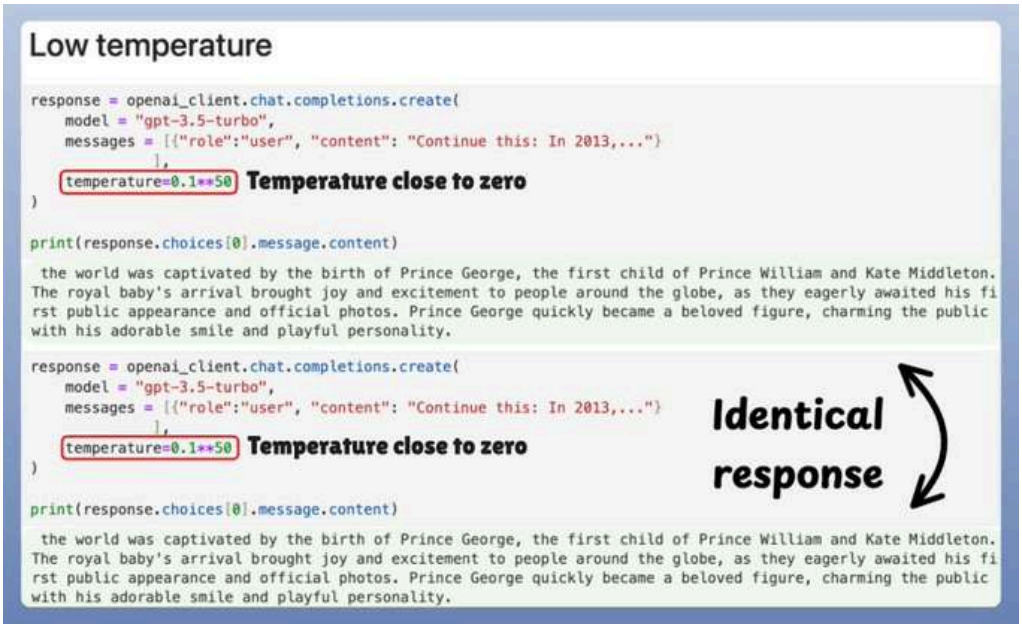


And the parameters of this distribution are the trained weights!

The training (or rather pre-training) is supervised.

But there is a problem!

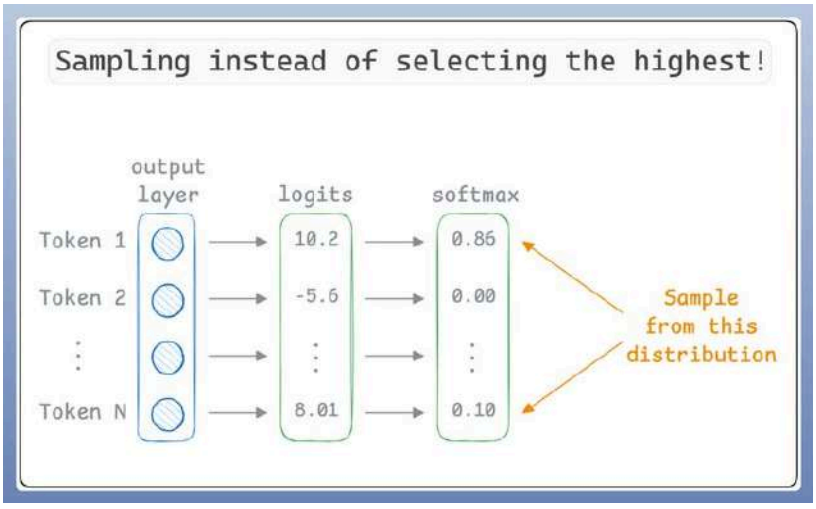
If we always pick the word with the highest probability, we end up with repetitive outputs, making LLMs almost useless and stifling their creativity.



This is where temperature comes into the picture.

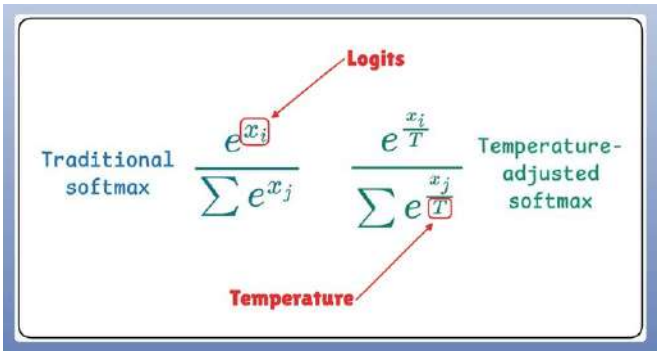
Let's understand what's going on..

To make LLMs more creative, instead of selecting the best token (for simplicity let's think of tokens as words), they "sample" the prediction.

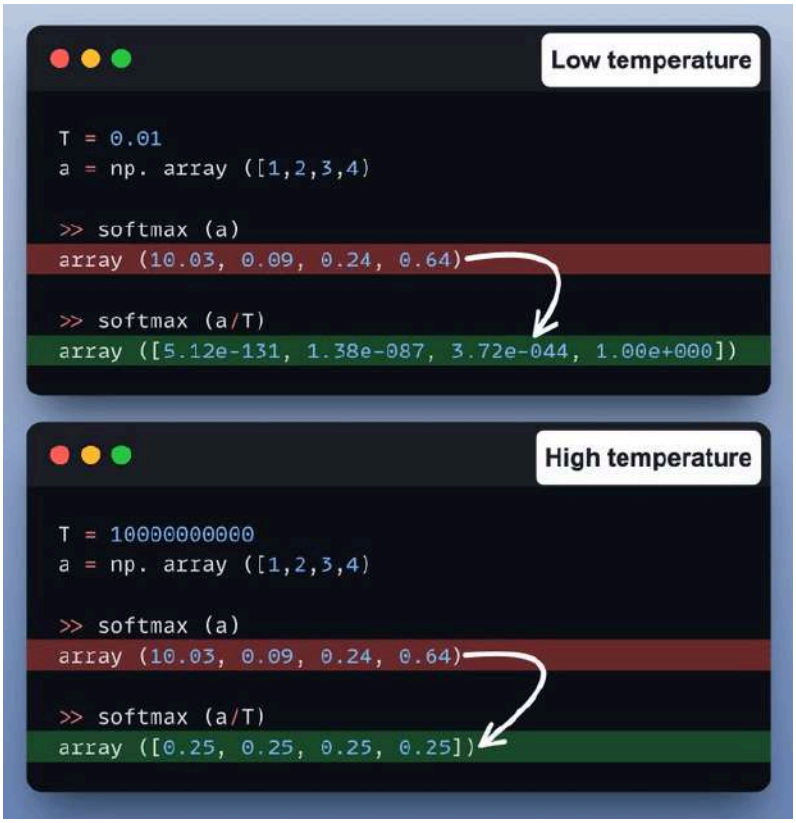


So even if “Token 1” has the highest score, it may not be chosen since we are sampling.

Now, temperature introduces the following tweak in the softmax function, which, in turn, influences the sampling process:



Let's take a code example!



- At low temperature, probabilities concentrate around the most likely token, resulting in nearly greedy generation.
- At high temperature, probabilities become more uniform, producing highly random and stochastic outputs.

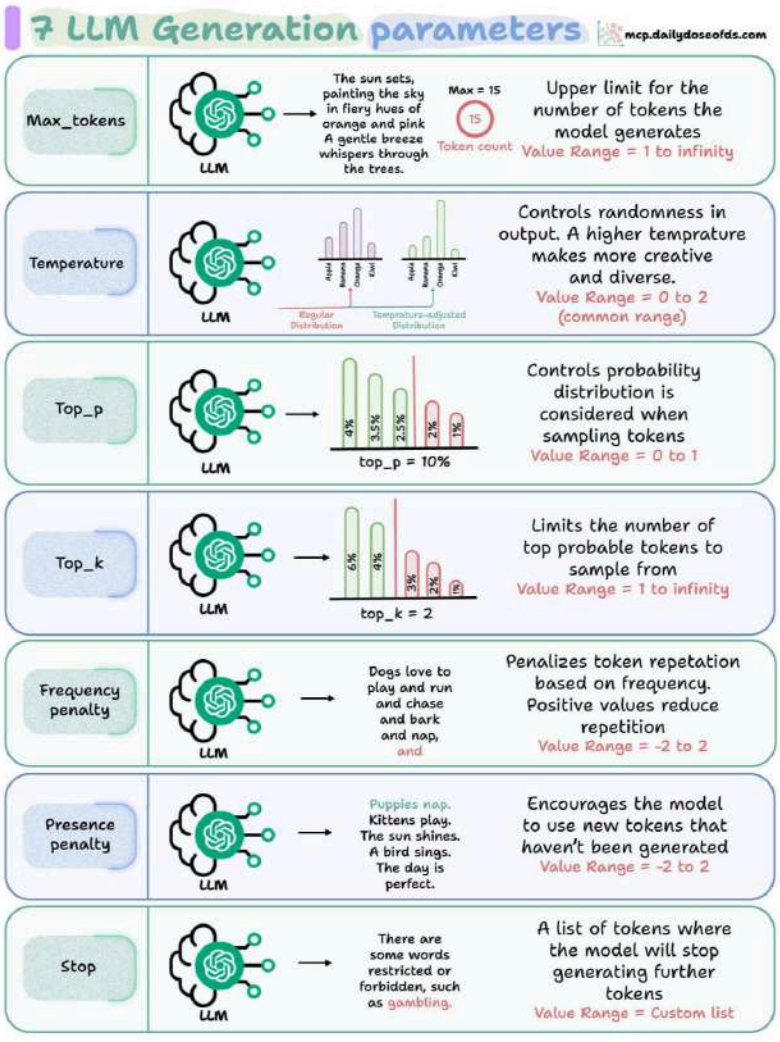
And that is how LLMs work and generate text.

7 LLM Generation Parameters

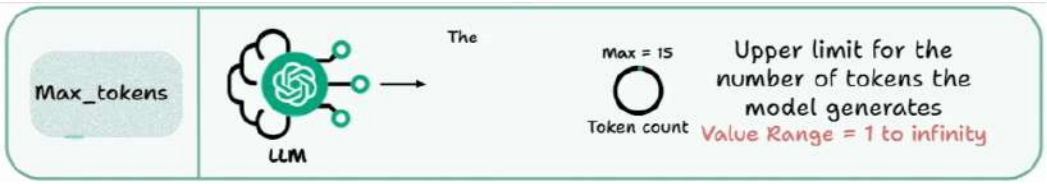
Every generation from an LLM is shaped by parameters under the hood.

Knowing how to tune is important so that you can produce sharp and more controlled outputs.

Here are the 7 levers that matter most:



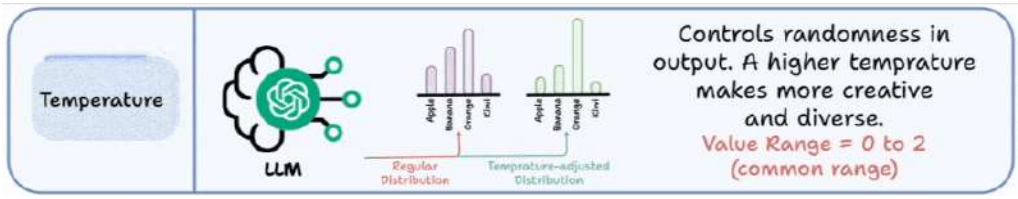
1) Max tokens



This is a hard cap on how many tokens the model can generate in one response.

Too low → truncated outputs; too high → could lead to wasted compute.

2) Temperature

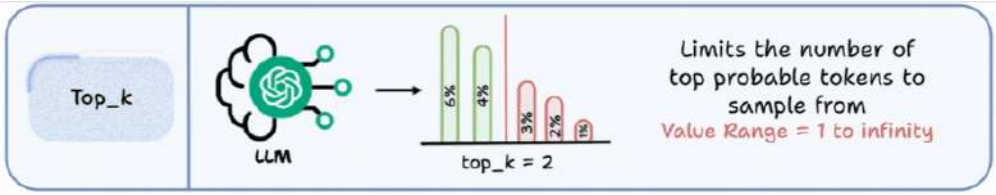


Governs randomness. Low temperature (~0) makes the model deterministic.

Higher temperature (0.7–1.0) boosts creativity, diversity, but also noise.

Use case: lower for QA/chatbots, higher for brainstorming/creative tasks.

3) Top-k



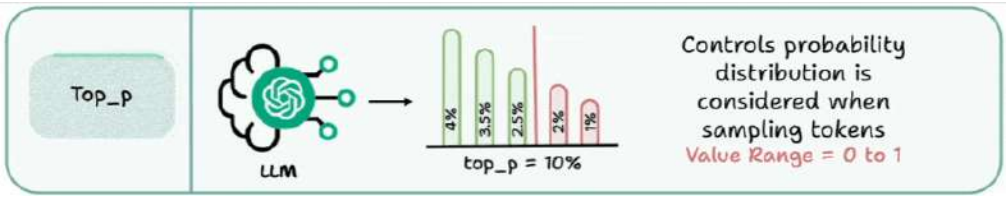
The default way to generate the next token is to sample from all tokens, proportional to their probability.

This parameter restricts sampling to the top k most probable tokens.

Example: k=5 → model only considers 5 most likely next tokens during sampling.

Helps enforce focus, but overly small k may give repetitive outputs.

4) Top-p (nucleus sampling)

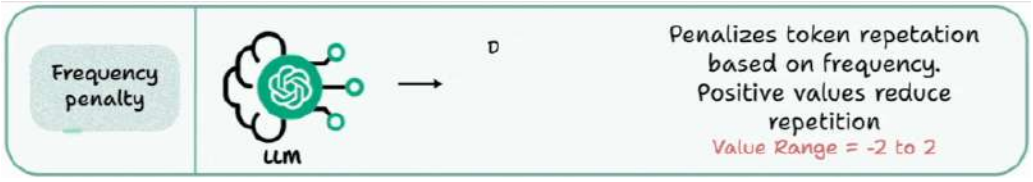


Instead of picking from all tokens or top k tokens, model samples from a probability mass up to p.

Example: top_p=0.9 → only the smallest set of tokens covering 90% probability are considered.

More adaptive than top_k, useful when balancing coherence with diversity.

5) Frequency penalty



Reduces likelihood of reusing tokens that have already appeared frequently.

Positive values discourage repetition, negative values exaggerate it.

Useful for summarization (avoid redundancy) or poetry (intentional repetition).

6) Presence penalty

Encourages the model to bring in new tokens not yet seen in the text.

Higher values push for novelty, lower values make the model stick to known patterns.

Handy for exploratory generation where diversity of ideas is valued.

7) Stop sequences

Custom list of tokens that immediately halt generation.

Critical in structured outputs (e.g., JSON), preventing spillover text.

Lets you enforce strict response boundaries without heavy prompt engineering.

Bonus: min-p sampling

min-p was introduced earlier this year. It's similar to top-p sampling, but instead of keeping a fixed cumulative probability mass (say, 90%), it dynamically adjusts based on the model's confidence.

It looks at the probability of the most likely token and only keeps tokens that are at least a certain fraction (say 10%) as likely.

For instance, if your top token has 60% probability, then only a few options remain. But if it's just 20%, many others can pass the threshold.

Basically, it automatically tightens or loosens the sampling pool depending on how confident the model is about the most likely token, which helps you achieve coherence when the model is confident and diversity when it's not that confident.

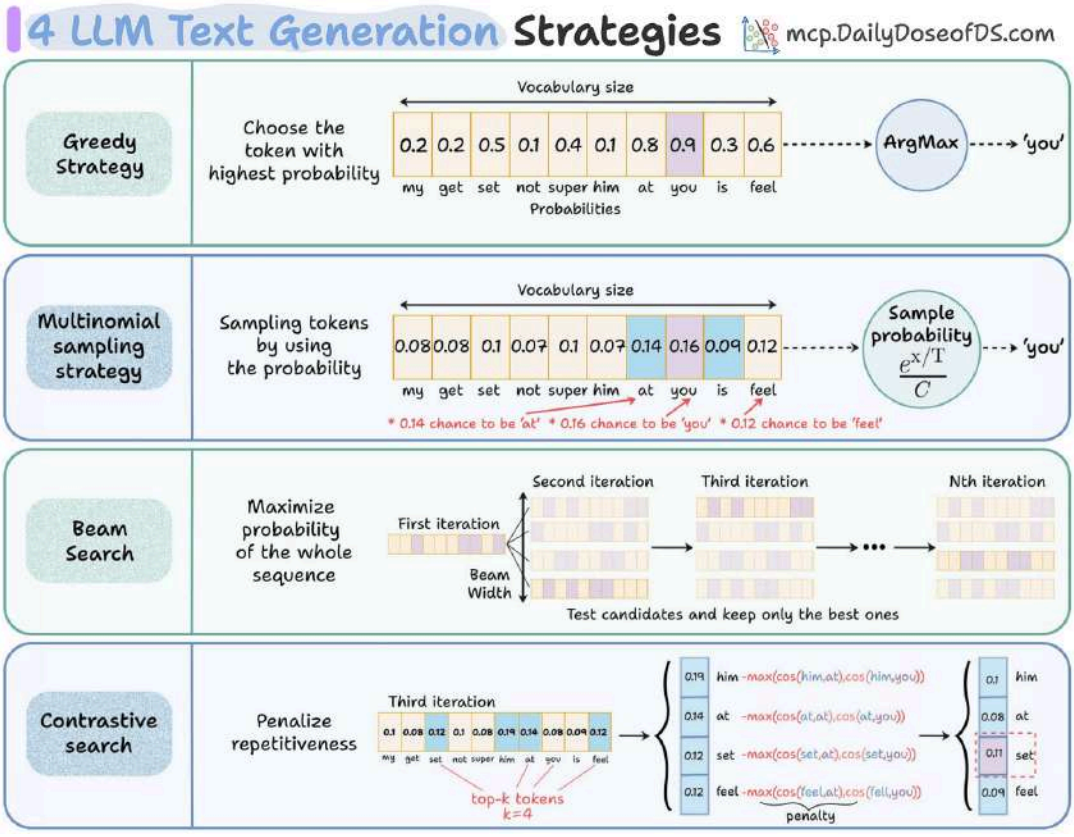
4 LLM Text Generation Strategies

Every time you prompt an LLM, it doesn't "know" the whole sentence in advance. Instead, it predicts the next token step by step.

But here's the catch: predicting probabilities is not enough. We still need a strategy to pick which token to use at each step.

And different strategies lead to very different styles of output.

Here are the 4 most common strategies for text generation:

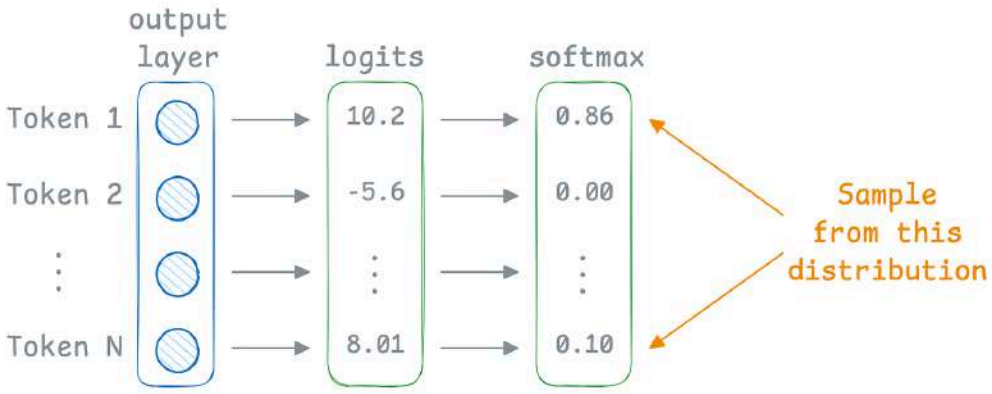


Approach 1: Greedy strategy

The naive approach greedily chooses the word with the highest probability from the probability vector, and autoregresses. This is often not ideal since it leads to repetitive sentences.

Approach 2: Multinomial sampling strategy

Instead of always picking the top token, we can sample from the probability distribution available in the probability vector.



The temperature parameter controls the randomness in the generation (covered in detail here).

Approach 3: Beam search

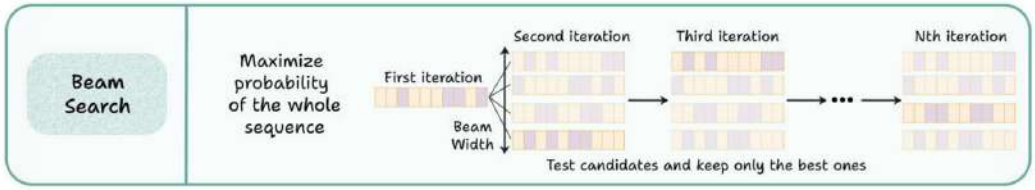
Both approach 1 and approach 2 have a problem. They only focus on the most immediate token to be generated. Ideally, we care about maximizing the probability of the whole sequence, not just the next token.

$$P(t_1, t_2, \ldots, t_N \mid \text{Prompt}) = \prod_{i=1}^{N} P(t_i \mid \text{Prompt}, t_1, \ldots, t_{i-1})$$

To maximize this product, you'd need to know future conditionals (what comes after each candidate).

But when decoding, we only know probabilities for the next step, not the downstream continuation.

Beam search tries to approximate the true global maximization:



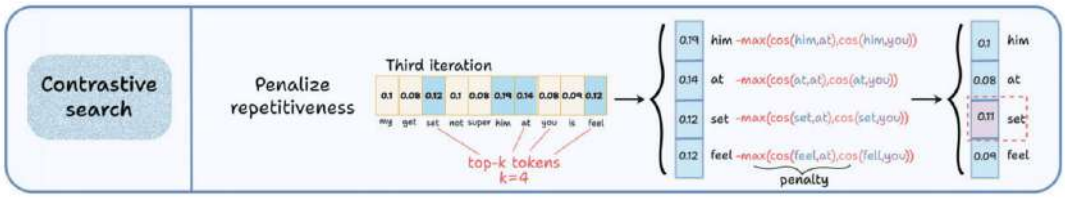
At each step, it expands the top k partial sequences (the beam).

Some beams may have started with less probable tokens initially, but lead to much higher-probability completions.

By keeping alternatives alive, beam search explores more of the probability tree.

This is widely used in tasks like machine translation, where correctness matters more than creativity.

Approach 4: Contrastive search



This is a newer method that balances fluency with diversity.

Essentially, it penalizes repetitive continuations by checking how similar a candidate token is to what's already been generated to have more diversity in the output.

At each step, the model considers candidate tokens.

Applies a penalty if the token is too similar to what's already been generated.

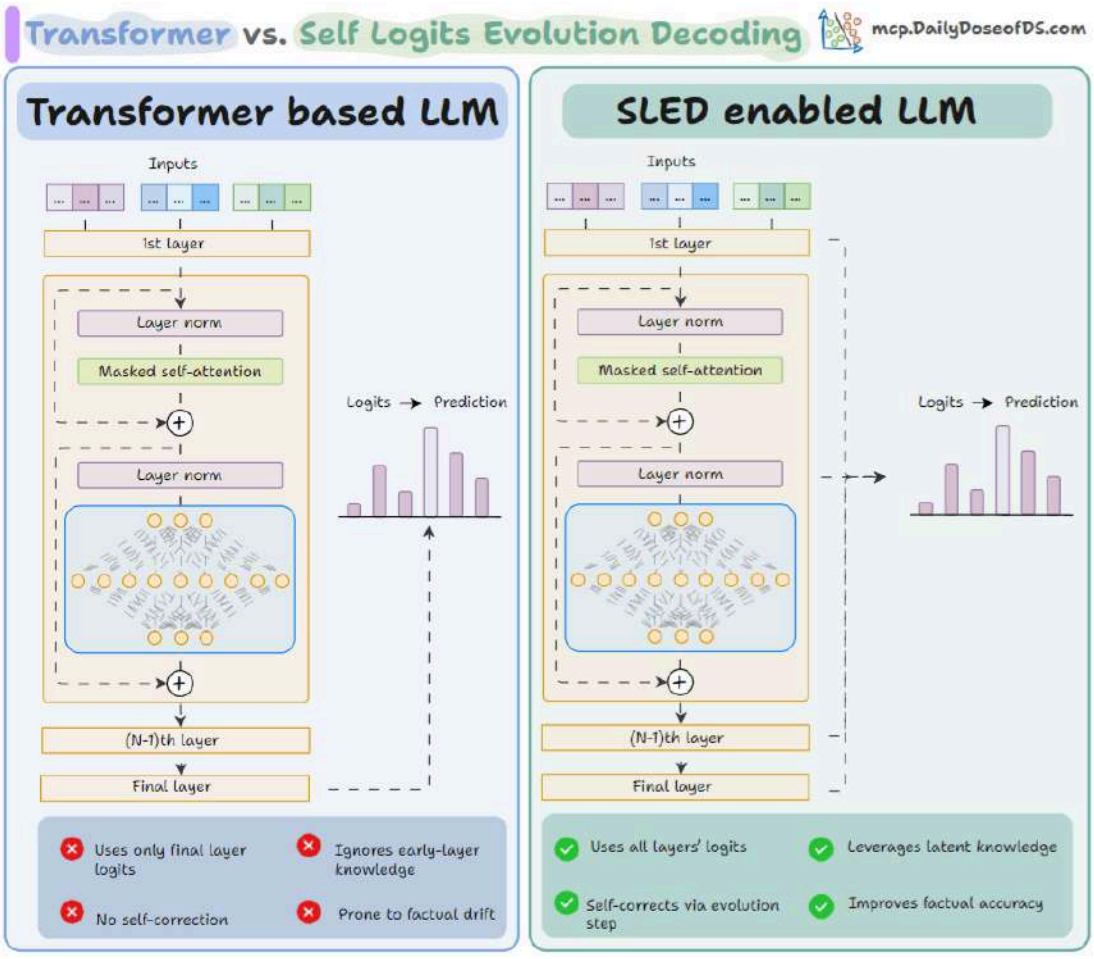
Selects the token that balances probability and diversity.

This way, it also prevents "stuck in a loop" problems while keeping coherence high.

It's especially effective for longer generations like stories, where repetition can easily creep in.

Bonus: SLED - Self-Logits Evolution Decoding

All the decoding strategies above rely on the logits produced by the final layer, which is how Transformers normally generate text. The issue is that factual signals present in earlier layers can fade as the model goes deeper, leading the final layer to favor fluent but occasionally inaccurate outputs.



SLED introduces a small but meaningful change: instead of using only the final layer's logits, it looks at how logits evolve across *all* layers. Each layer contributes its own prediction, and SLED measures how closely these predictions agree. It then nudges the final logits toward this layer-wise consensus before selecting the next token.

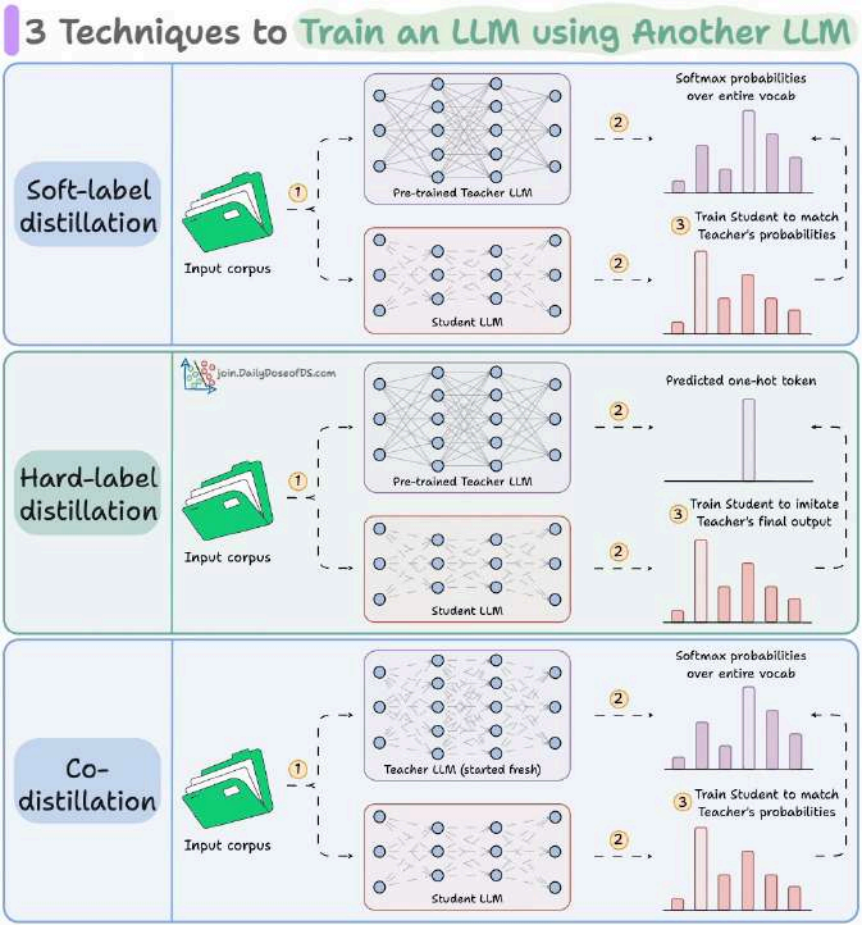
This requires no retraining, extra data or additional compute. By leveraging the model's full internal knowledge rather than only its last step, SLED produces more grounded and factual generations with the same architecture.

3 Techniques to Train An LLM Using Another LLM

LLMs don't just learn from raw text; they also learn from each other:

- Llama 4 Scout and Maverick were trained using Llama 4 Behemoth.
- Gemma 2 and 3 were trained using Google's proprietary Gemini.

Distillation helps us do so, and the visual below depicts three popular techniques.



The idea is to transfer "knowledge" from one LLM to another, which has been quite common in traditional deep learning

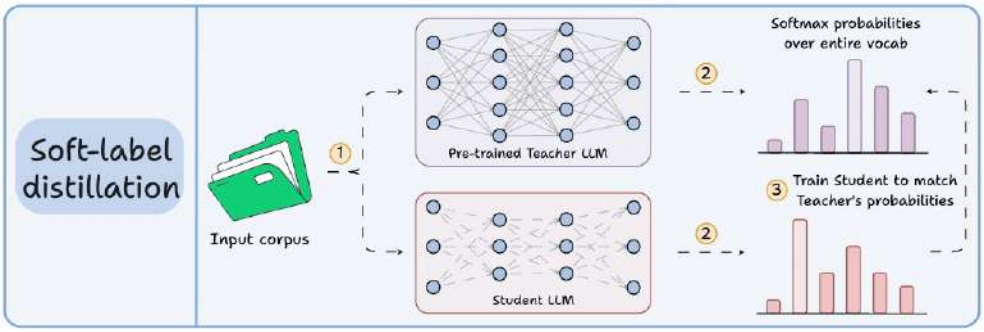
Distillation in LLMs can happen at two stages:

- Pre-training
  - Train the bigger Teacher LLM and the smaller student LLM together.
  - Llama 4 did this.
- Post-training:
  - Train the bigger Teacher LLM first and distill its knowledge to the smaller student LLM.
  - DeepSeek did this by distilling DeepSeek-R1 into Qwen and Llama 3.1 models.

You can also apply distillation during both stages, which Gemma 3 did.

Here are the three commonly used distillation techniques:

1) Soft-label distillation



- Use a fixed pre-trained Teacher LLM to generate softmax probabilities over the entire corpus.
- Pass this data through the untrained Student LLM as well to get its softmax probabilities.
- Train the Student LLM to match the Teacher's probabilities.

Visibility over the Teacher's probabilities ensures maximum knowledge (or reasoning) transfer.

However, you must have access to the Teacher's weights to get the output probability distribution.

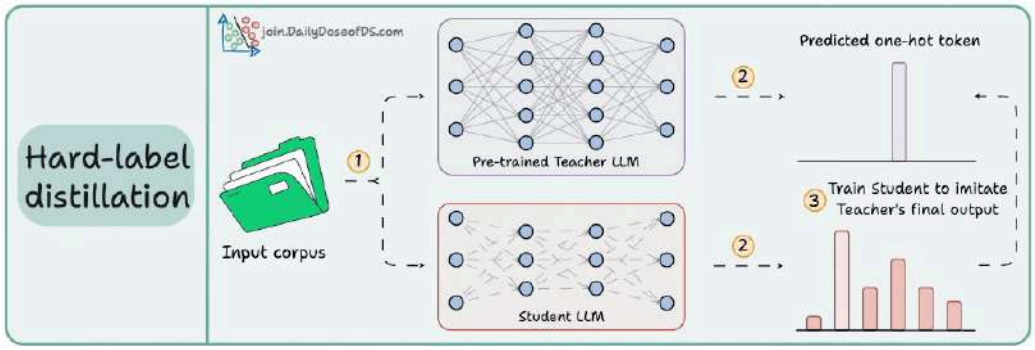
Even if you have access, there's another problem!

Say your vocab size is 100k tokens and your data corpus is 5 trillion tokens.

Since we generate softmax probabilities of each input token over the entire vocabulary, you would need 500 million GBs of memory to store soft labels under float8 precision.

The second technique solves this.

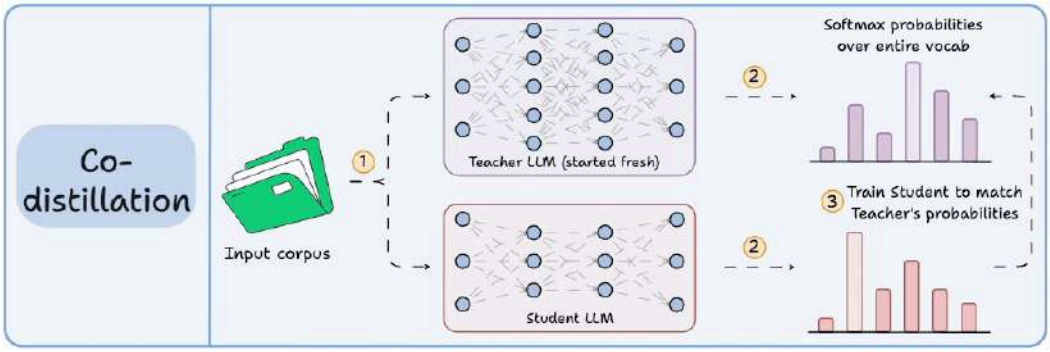
2) Hard-label distillation



- Use a fixed pre-trained Teacher LLM to just get the final one-hot output token.
- Use the untrained Student LLM to get the softmax probabilities from the same data.
- Train the Student LLM to match the Teacher's probabilities.

DeepSeek did this by distilling DeepSeek-R1 into Qwen and Llama 3.1 models.

3) Co-distillation



- Start with an untrained Teacher LLM and an untrained Student LLM.
- Generate softmax probabilities over the current batch from both models.
- Train the Teacher LLM as usual on the hard labels.
- Train the Student LLM to match its softmax probabilities to those of the Teacher.

Llama 4 did this to train Llama 4 Scout and Maverick from Llama 4 Behemoth.

Of course, during the initial stages, soft labels of the Teacher LLM won't be accurate.

That is why Student LLM is trained using both soft labels + ground-truth hard labels.

4 Ways to Run LLMs Locally

Being able to run LLMs locally has many upsides:

- Privacy since your data never leaves your machine.
- Testing things locally before moving to the cloud and more.

Let's discuss the four ways to run LLMs locally.

1) Ollama

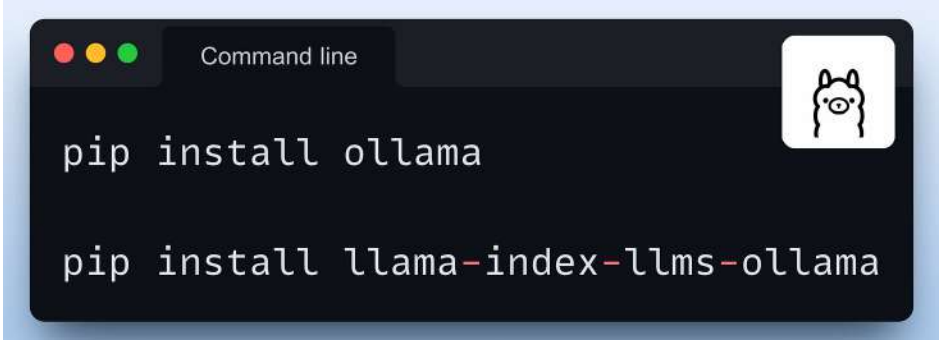
Running a model through Ollama is as simple as executing this command:

To get started, install Ollama with a single command:

Done!

Now, you can download any of the supported models using these commands:

For programmatic usage, you can also install the Python package of Ollama or its integration with orchestration frameworks like Llama Index or CrewAI:



2) LMStudio

LMStudio can be installed as an app on your computer.

The app does not collect data or monitor your actions. Your data stays local on your machine. It's free for personal use.

It offers a ChatGPT-like interface, allowing you to load and eject models as you chat. This video shows its usage:

Just like Ollama, LMStudio supports several LLMs as well.

3) vLLM

vLLM is a fast and easy-to-use library for LLM inference and serving(*more details in LLM deployment section*)

With just a few lines of code, you can locally run LLMs (like DeepSeek) in an OpenAI-compatible format:

```
pip install vllm

vllm serve deepseek-ai/DeepSeek-R1-Distill-Qwen-1.5B \
    --enable-reasoning --reasoning-parser deepseek_r1
```

```
from openai import OpenAI

# Modify OpenAI's API key and API base to use vLLM's API server.
openai_api_key = "EMPTY"
openai_api_base = "http://localhost:8000/v1"

client = OpenAI(
    api_key=openai_api_key,
    base_url=openai_api_base,
)

models = client.models.list()
model = models.data[0].id

# Round 1
messages = [{"role": "user", "content": "9.11 and 9.8, which is greater?"}]
response = client.chat.completions.create(model=model, messages=messages)

reasoning_content = response.choices[0].message.reasoning_content
content = response.choices[0].message.content

print("reasoning_content:", reasoning_content)
print("content:", content)
```

4) LlamaCPP

LlamaCPP enables LLM inference with minimal setup and good performance.

```
brew install llama. cpp

# increase VRAM limit
sudo sysctl iogpu.wired_limit_mb=180000
# downloads ~150 GB, requires ~180 GB VRAM

llama-server -c 8192 -ub 64 \
--model-url https://huggingface.co/unsloth/DeepSeek-R1-
GGUF/resolve/main/DeepSeek-R1-UD-IQ1_S/DeepSeek-R1-UD-IQ1_S-00001-
of-00003.gguf

# open http://127.0.0.1:8080
```

Transformer vs. Mixture of Experts in LLMs

Up to this point, we've explored how LLMs are built, trained and how they generate text.

All modern LLMs rely on the Transformer architecture, but there is another important question:

How do we scale models further without making them impossibly large and expensive to run?

This is where Mixture of Experts (MoE) architectures come in.

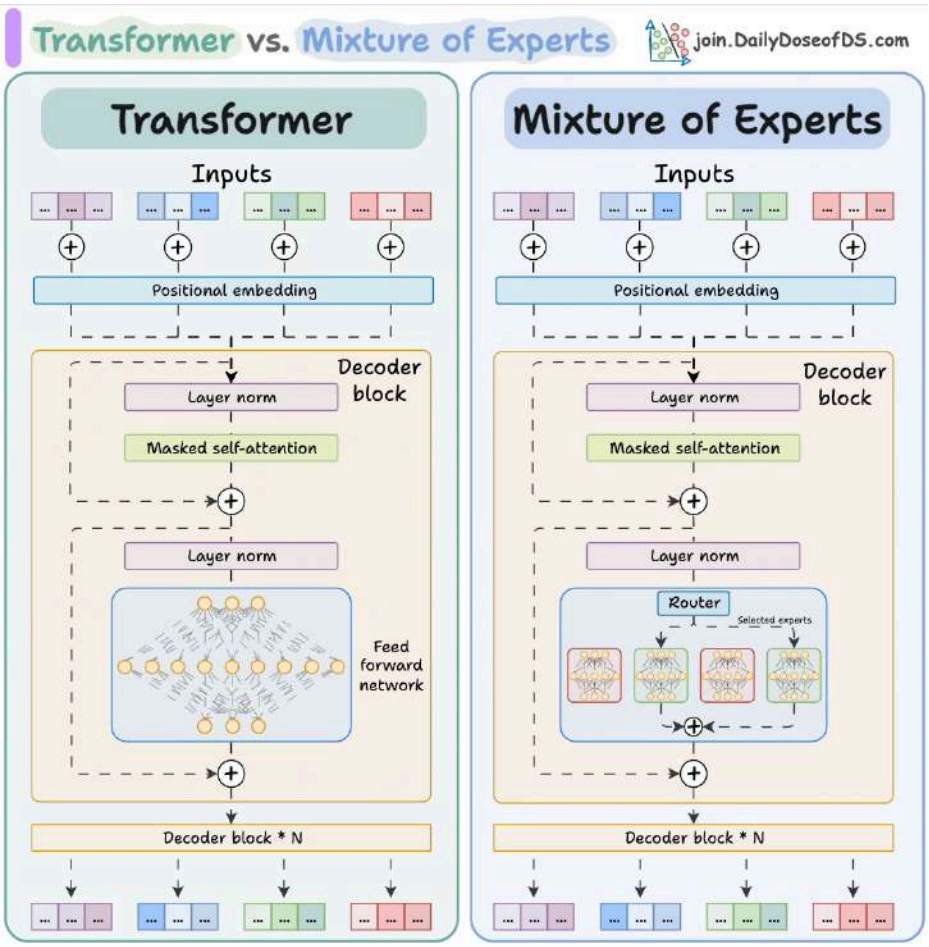
MoE keeps the overall parameter count large but activates only a small subset of "experts" for each token.

This allows models to grow in capacity without proportional increases in compute.

With that context, let's compare the traditional Transformer block with the MoE alternative.

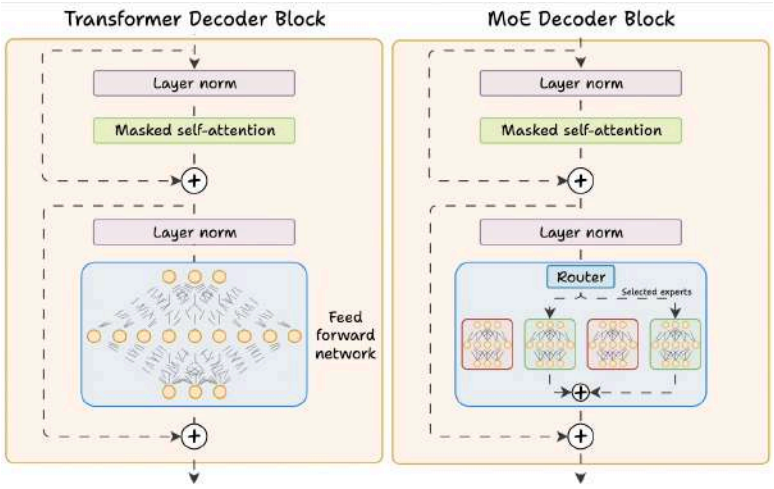
Mixture of Experts (MoE) is a popular architecture that uses different "experts" to improve Transformer models.

The visual below explains how they differ from Transformers.



Let's dive in to learn more about MoE!

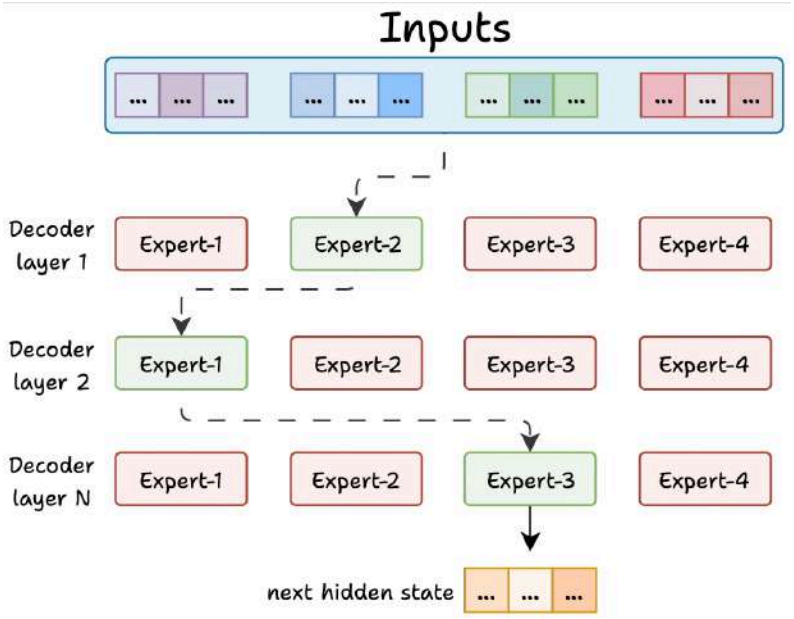
Transformer and MoE differ in the decoder block:



- Transformer uses a feed-forward network.
- MoE uses experts, which are feed-forward networks but smaller compared to that in Transformer.

During inference, a subset of experts are selected. This makes inference faster in MoE.

Also, since the network has multiple decoder layers:



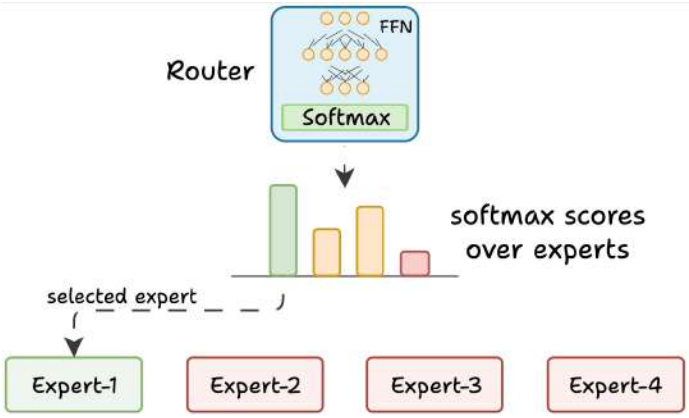
- the text passes through different experts across layers.
- the chosen experts also differ between tokens.

But how does the model decide which experts should be ideal?

The router does that.

The router is like a multi-class classifier that produces softmax scores over experts. Based on the scores, we select the top K experts.

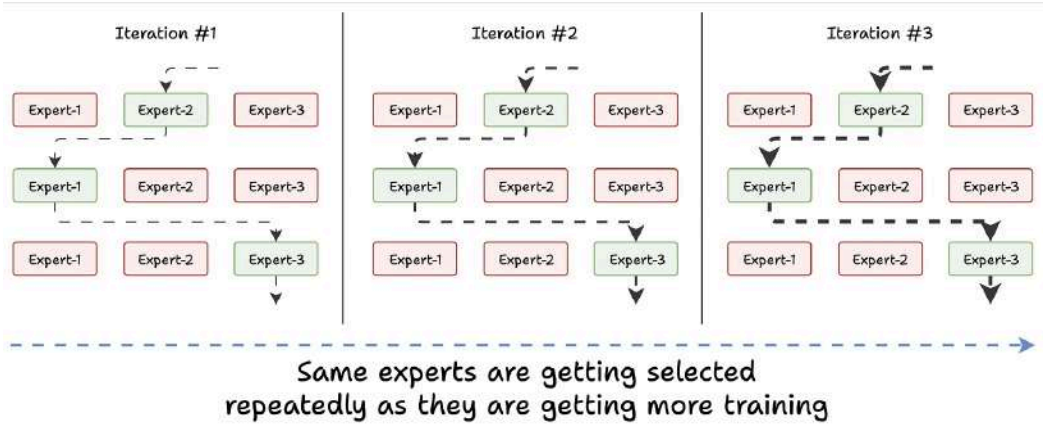
The router is trained with the network and it learns to select the best experts.



But it isn't straightforward.

There are challenges.

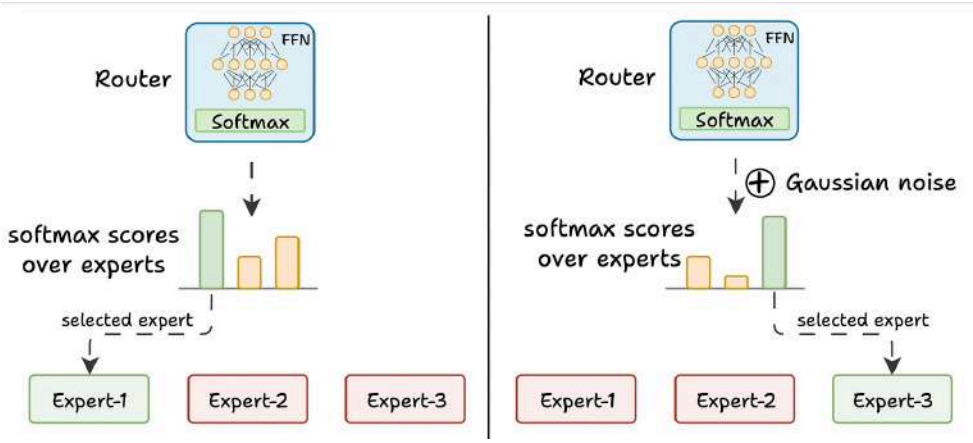
Challenge 1) Notice this pattern at the start of training:



- The model selects "Expert 2" (randomly since all experts are similar).
- The selected expert gets a bit better.
- It may get selected again since it's the best.
- This expert learns more.
- The same expert can get selected again since it's the best.
- It learns even more.
- And so on!

Essentially, this way, many experts go under-trained!

We solve this in two steps:



- Add noise to the feed-forward output of the router so that other experts can get higher logits.
- Set all but top K logits to -infinity. After softmax, these scores become zero.

This way, other experts also get the opportunity to train.

Challenge 2) Some experts may get exposed to more tokens than others - leading to under-trained experts.

We prevent this by limiting the number of tokens an expert can process.

If an expert reaches the limit, the input token is passed to the next best expert instead.

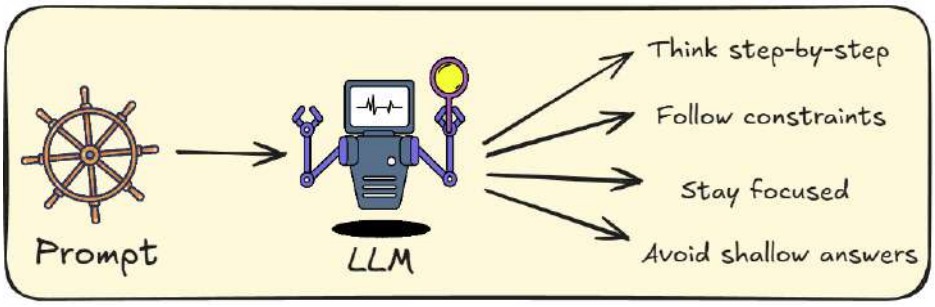
MoEs have more parameters to load. However, a fraction of them are activated since we only select some experts.

This leads to faster inference. Mixtral 8x7B by MistralAI is one famous LLM that is based on MoE.

Prompt Engineering

What is Prompt Engineering?

LLMs are powerful, but they don't automatically know what you want. Prompt engineering is the simplest way to control them.



Think of it as the steering wheel for the LLM.

Small adjustments completely shift the direction of the output.

You're not changing weights (the learned parameters inside the model). You're changing instructions and that changes everything.

A good prompt helps the model:

- Think step-by-step
- Follow constraints
- Stay focused
- Avoid shallow answers

It's the fastest, lowest-effort way to get better results from any model.

Let's explore three prompting techniques that significantly improve an LLM's reasoning ability.

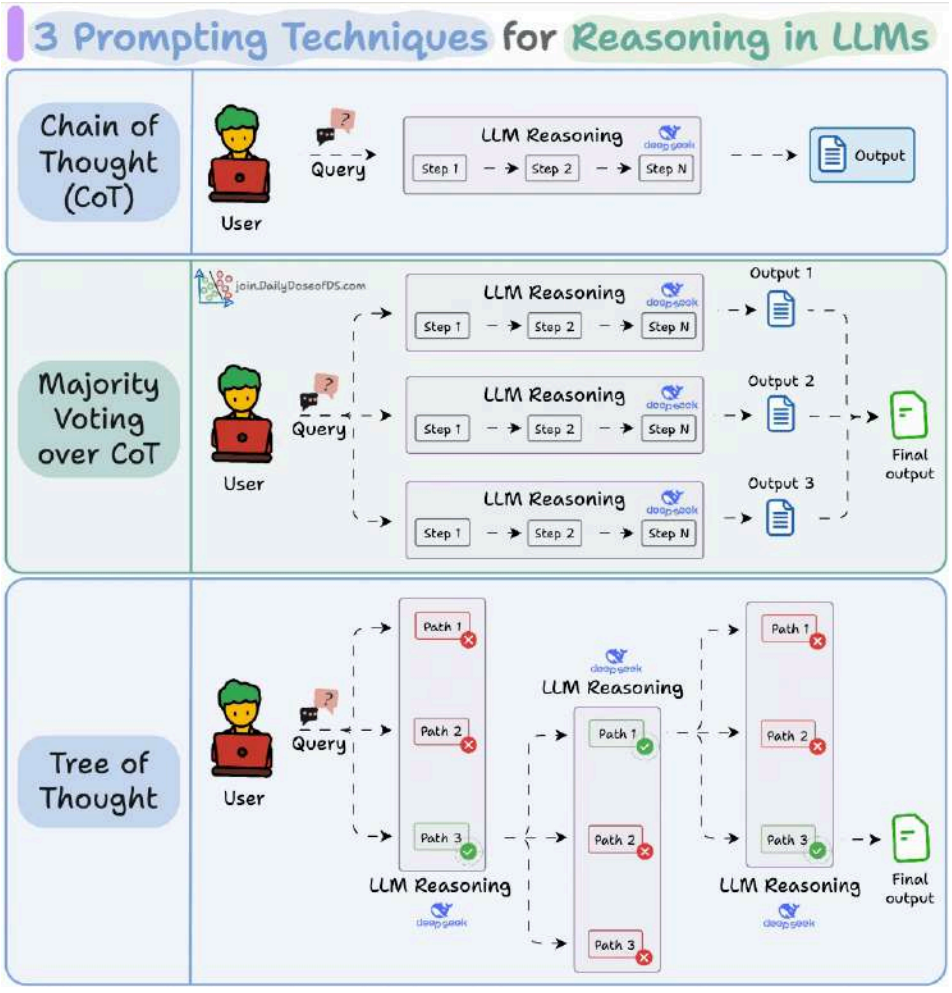
3 prompting techniques for reasoning in LLMs

A large part of what makes such tools so powerful isn't just their ability to write code, but their ability to reason through it.

And that's not unique to code. It's the same when we prompt LLMs to solve complex reasoning tasks like math, logic, or multi-step problems.

Let's look at three popular prompting techniques that help LLMs think more clearly before they answer.

These are depicted below:



1) Chain of Thought (CoT)

The simplest and most widely used technique.

Instead of asking the LLM to jump straight to the answer, we nudge it to reason step by step.

This often improves accuracy because the model can walk through its logic before committing to a final output.

For instance:

```
Q: If John has 3 apples and gives away 1, how many are left?

Let's think step by step:
```

It's a simple example but this tiny nudge can unlock reasoning capabilities that standard zero-shot prompting could miss.

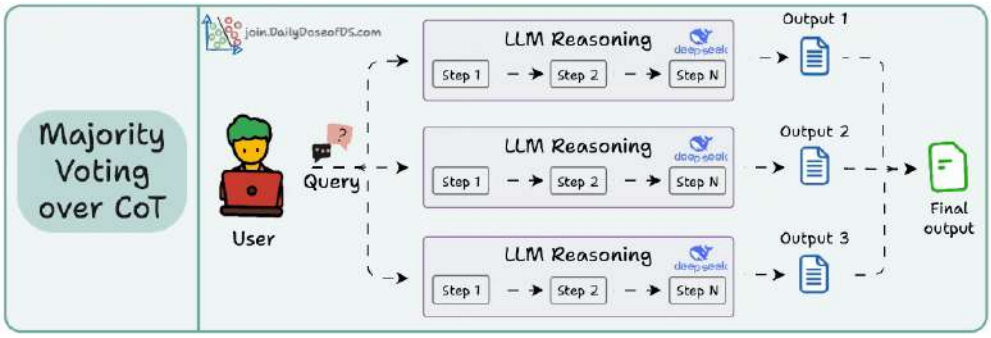
2) Self-Consistency (a.k.a. Majority Voting over CoT)

CoT is useful but not always consistent.

If you prompt the same question multiple times, you might get different answers depending on the temperature setting (we covered temperature in LLMs here).

Self-Consistency embraces this variation.

You ask the LLM to generate multiple reasoning paths and then select the most common final answer.



It's a simple idea: when in doubt, ask the model several times and trust the majority.

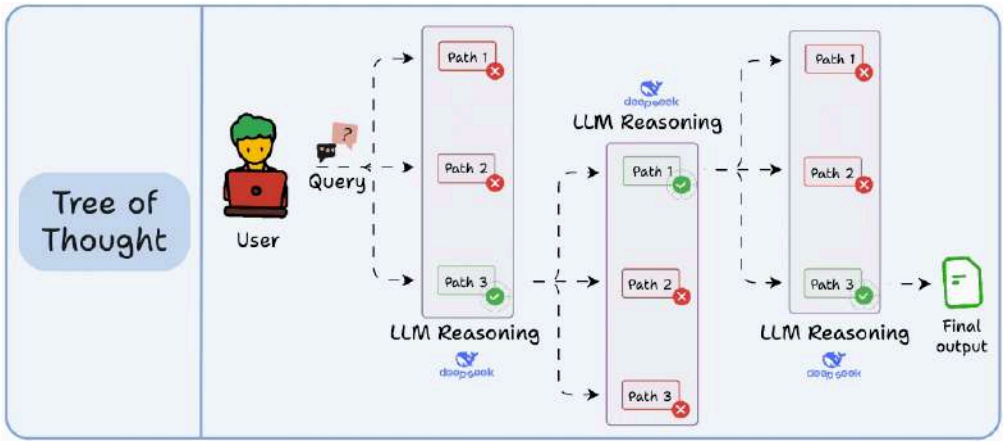
This technique often leads to more robust results, especially on ambiguous or complex tasks.

However, it doesn't evaluate how the reasoning was done—just whether the final answer is consistent across paths.

3) Tree of Thoughts (ToT)

While Self-Consistency varies the final answer, Tree of Thoughts varies the steps of reasoning at each point and then picks the best path overall.

At every reasoning step, the model explores multiple possible directions. These branches form a tree, and a separate process evaluates which path seems the most promising at a particular timestamp.



Think of it like a search algorithm over reasoning paths, where we try to find the most logical and coherent trail to the solution.

It's more compute-intensive, but in most cases, it significantly outperforms basic CoT.

CoT, Self-Consistency, and ToT all improve how the model reasons through a problem.

But they still rely on free-form thinking, which breaks down in long, rule-heavy tasks.

That's where ARQ comes in.

Bonus: ARQ

Here's the core problem with current techniques that this new approach solves.

We have enough research to conclude that LLMs often struggle to assess what truly matters in a particular stage of a long, multi-turn conversation.

For instance, when you give Agents a 2,000-word system prompt filled with policies, tone rules, and behavioral dos and don'ts, you expect them to follow it word by word.

- But here's what actually happens:
- They start strong initially.
- Soon, they drift and start hallucinating.
- Shortly after, they forget what was said five turns ago.

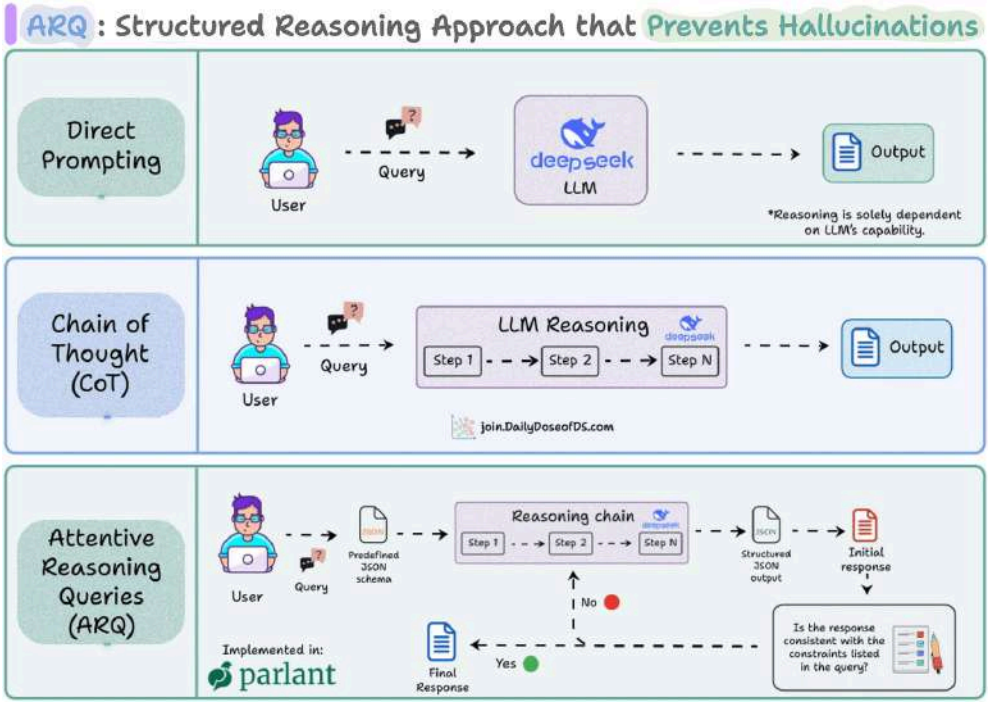
And finally, the LLM that was supposed to "never promise a refund" is happily offering one.

This means they can easily ignore crucial rules (stated initially) halfway through the process.

We expect techniques like Chain-of-Thought will help.

But even with methods like CoT, reasoning remains free-form, i.e., the model "thinks aloud" but it has limited domain-specific control.

That's the exact problem the new technique, called Attentive Reasoning Queries (ARQs), solves.



Instead of letting LLMs reason freely, ARQs guide them through explicit, domain-specific questions.

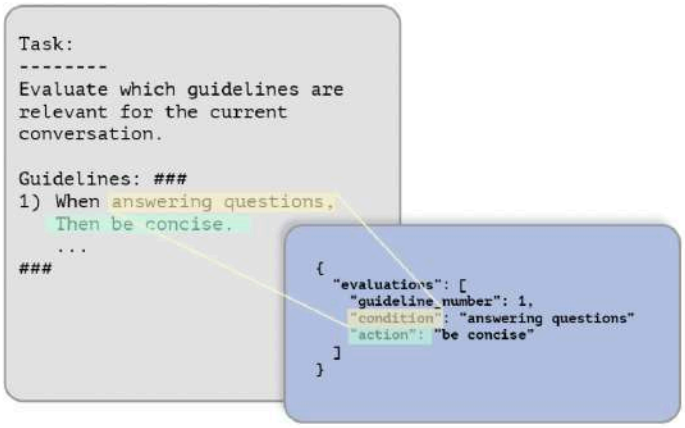
Essentially, each reasoning step is encoded as a targeted query inside a JSON schema.

For example, before making a recommendation or deciding on a tool call, the LLM is prompted to fill structured keys like:

```
{
  “current_context”: “Customer asking about refund eligibility”,
  “active_guideline”: “Always verify order before issuing refund”,
  “action_taken_before”: false,
  “requires_tool”: true,
  “next_step”: “Run check_order_status()”
}
```

This type of query does two things:

1. Reinstate critical instructions by keeping the LLM aligned mid-conversation.
2. Facilitate intermediate reasoning, so that the decisions are auditable and verifiable.



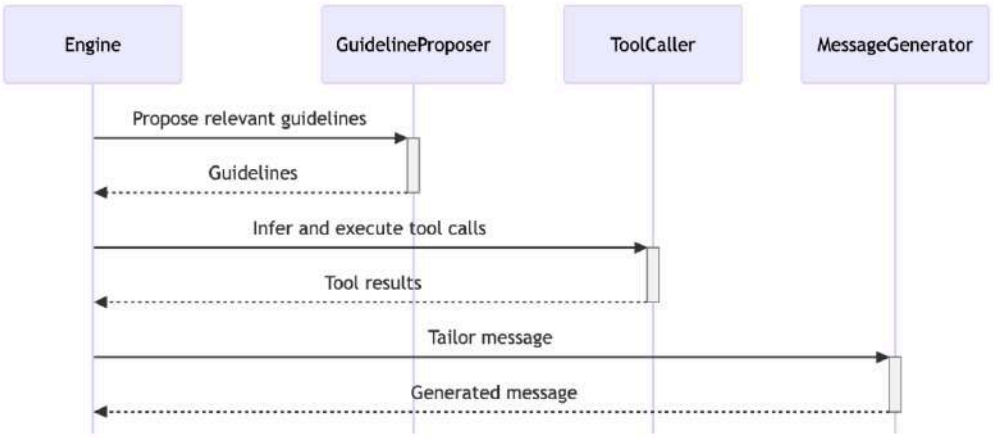
By the time the LLM generates the final response, it’s already walked through a sequence of *controlled* reasoning steps, which did not involve any free text exploration (unlike techniques like CoT or ToT).

Here’s the success rate across 87 test scenarios:

- ARQ - 90.2%
- CoT reasoning - 86.1%
- Direct response generation - 81.5%

This approach is actually implemented in Parlant, a recently trending open-source framework to build instruction-following Agents.

ARQs are integrated into three key modules:



- Guideline proposer to decide which behavioral rules apply.
- Tool caller to determine what external functions to use.
- Message generator, when it produces the final customer-facing reply.

The core insight applies regardless of what tools you use:

When you make reasoning explicit, measurable, and domain-aware, LLMs stop improvising and start reasoning with intention. Free-form thinking sounds powerful, but in high-stakes or multi-turn scenarios, structure always wins.

ARQ solves the problem of uncontrolled reasoning by adding structure.

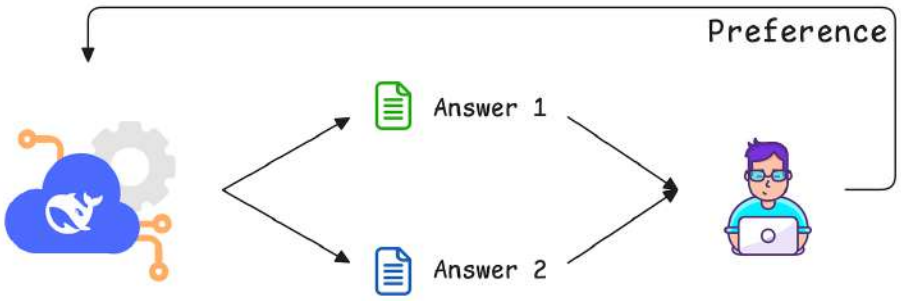
But there's another challenge: many aligned LLMs stop exploring alternative answers altogether.

Even with good reasoning steps, the model may collapse into the same safe, typical responses.

To regain that lost diversity without retraining the model, we use Verbalized Sampling.

Verbalized Sampling

Post-training alignment methods, such as RLHF, are designed to make LLMs helpful and safe.



However, these methods unintentionally cause a significant drop in output diversity (called mode collapse).

When an LLM collapses to a mode, it starts favoring a narrow set of predictable or stereotypical responses over other outputs.

According to a paper, mode collapse happens because the human preference data used to train the LLM has a hidden flaw called typicality bias.

> 3.1 TYPICALITY BIAS IN PREFERENCE DATA: COGNITIVE & EMPIRICAL EVIDENCE
>
> **Typicality Bias Hypothesis.** Cognitive psychology shows that people prefer text that is *familiar*, *fluent*, and *predictable*. This preference is rooted in various principles. For instance, the *mere-exposure effect* (Zajonc, 1968; Bornstein, 1989) and *availability heuristic* (Tversky & Kahneman, 1973) imply that frequent or easily recalled content feels more likely and is liked more. *Processing fluency* (Alter & Oppenheimer, 2009; Reber et al., 2004) suggests that easy-to-process content is automatically perceived as more truthful and higher quality. Moreover, *schema congruity* theory (Mandler, 2014; Meyers-Levy & Tybout, 1989) predicts that information that aligns with existing mental models will be accepted with less critical thought. We therefore hypothesize that these cognitive tendencies lead to a *typicality bias* in preference data, in which annotators systematically favor conventional text.

Here's how this happens:

Annotators are asked to rate different responses from an LLM, and later, the LLM is trained using a reward model that learns to mimic these human preferences.

However, it is observed that annotators naturally tend to favor answers that are more familiar, easy to read, and predictable. This is the typicality bias. So even if a new, creative answer is just as good (or correct) as a common one, the human's preference often leans toward the common one.

Due to this, the reward model boosts responses that the original (pre-aligned) model already considered likely.

This aggressively sharpens the LLM's probability distribution, collapsing the model's creative output to one or two dominant, highly predictable responses.

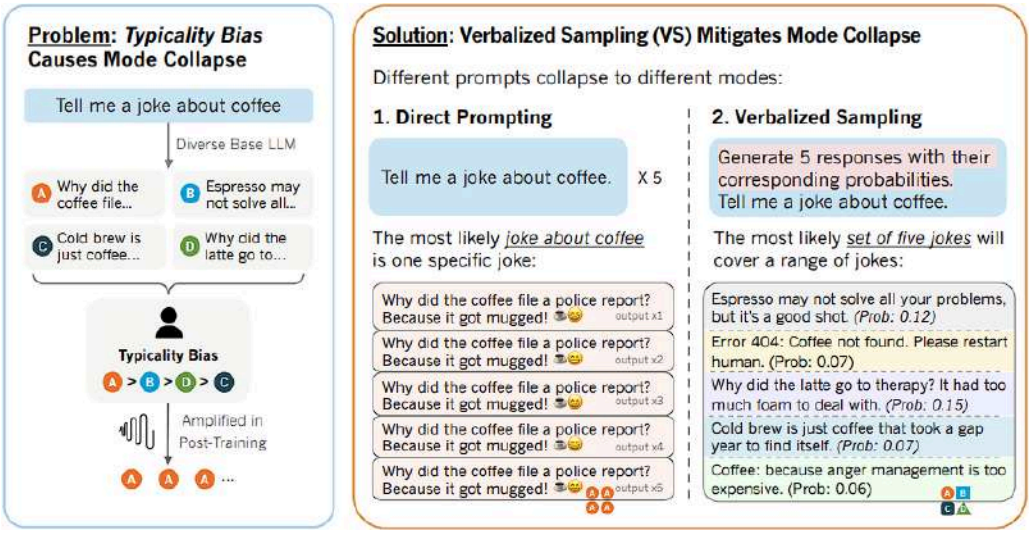
That said, this is not an irreversible effect, and the LLM still has two personalities after alignment:

The original model that learned the rich possibilities during pre-training.

The safety-focused, post-aligned model [to mention again, due to typicality bias, it had been unintentionally suppressed to strongly favor the most predictable response]

Verbalized sampling (VS) solves this.

It is a training-free prompting strategy introduced to circumvent mode collapse and recover the diverse distribution learned during pre-training.



The core idea of verbalized sampling is that the prompt itself acts like a mental switch.

When you directly prompt "Tell me a joke", the aligned personality immediately takes over and outputs the most reinforced answer.

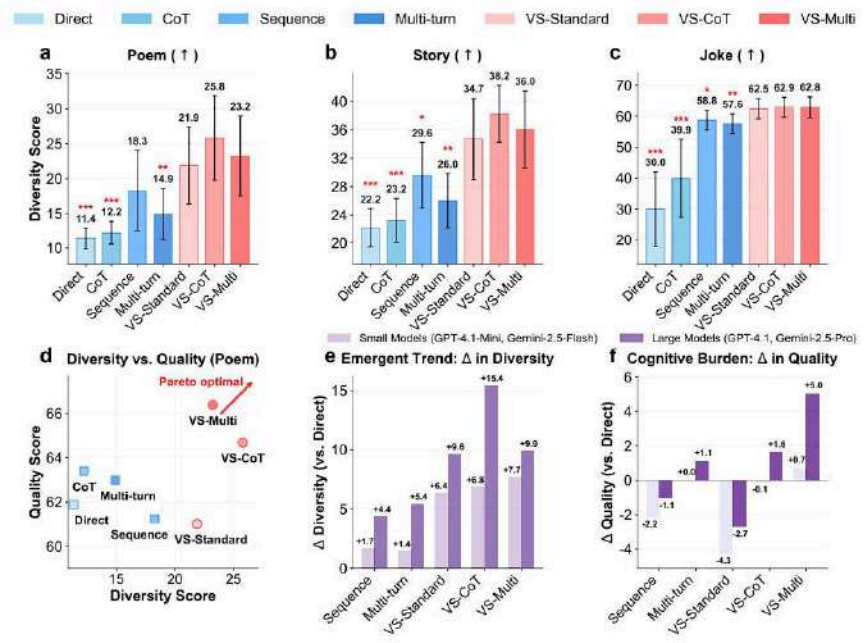
But in verbalized sampling, you prompt it with "Generate 5 responses with their corresponding probabilities. Tell me a joke."

In this case, the prompt does not request an instance, but a distribution.

This causes the aligned model to talk about its full knowledge and is forced to utilize the diverse distribution it learned during pre-training.

So essentially, by asking the LLM to verbalize the probability distribution, the model is able to tap into the broader, diverse set of ideas, which comes from the rich distribution that still exists inside its core pre-trained weights.

Experiments across various tasks demonstrate significant benefits:



Verbalized sampling significantly enhances diversity by 1.6-2.1x over direct prompting, while maintaining or improving quality. Variants like verbalized

sampling-based CoT (Chain-of-Thought) and verbalized sampling-based Multi improve generation diversity even further.

Larger, more capable models like GPT-4.1 and Gemini-2.5-Pro benefit more from verbalized sampling, showing diversity gains up to 2 times greater than smaller models.

Verbalized sampling better retains diversity across post-training stages (SFT, DPO, RLVR), recovering about 66.8% of the base model's original diversity, compared to a much lower retention rate for direct prompting.

Verbalized sampling's gains are independent of other methods, meaning it can be combined with techniques like temperature scaling, top-p sampling to achieve further improvements in the diversity-quality trade-off.

JSON prompting for LLMs

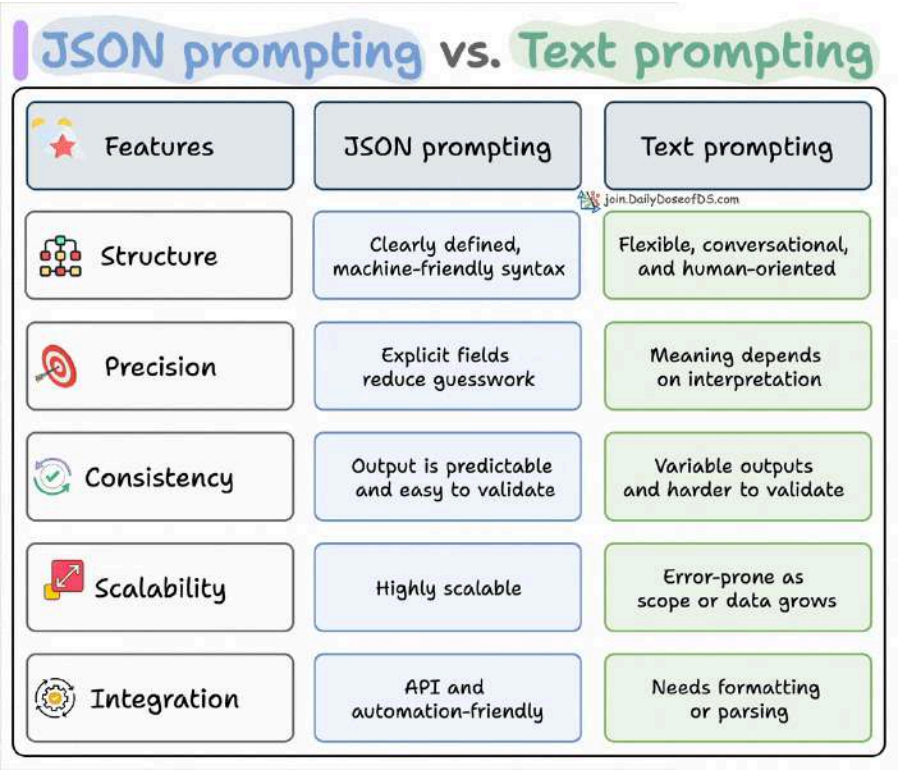
When you give an LLM an open-ended instruction, it has to guess what "good output" looks like.
Sometimes it adds extra commentary, sometimes it skips details, sometimes the formatting changes for no reason.

The problem isn't the model - it's the lack of structure in the prompt.

For tasks like extraction, reporting, automation, or analysis, you need the output to stay consistent every single time.

That's where JSON prompting helps.

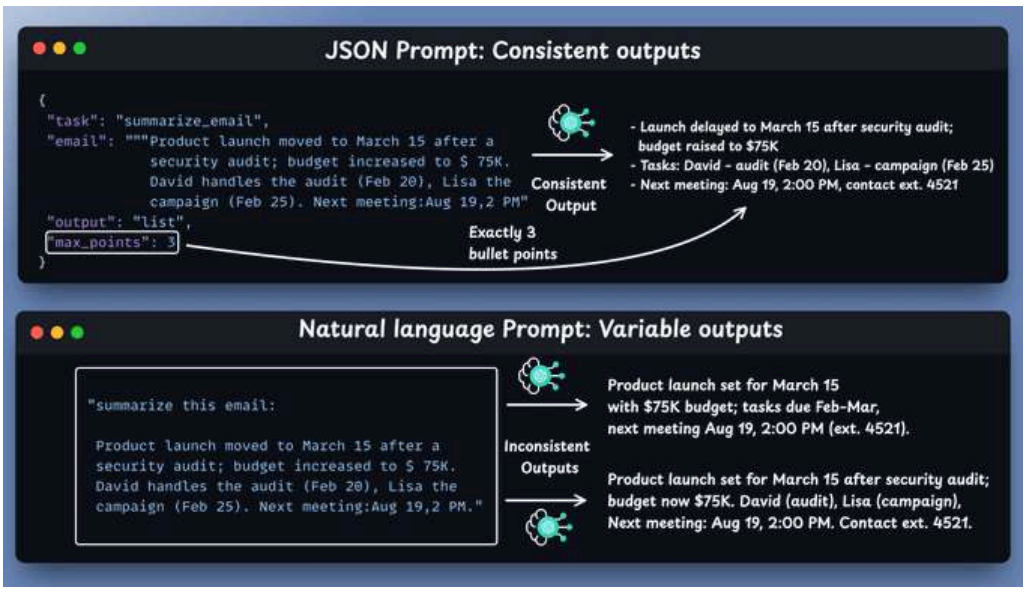
Let us discuss exactly what JSON prompting is and how it can drastically improve your AI outputs!



Natural language is powerful yet vague.

When you give instructions like "summarize this email" or "give me key takeaways," you leave room for interpretation, which can lead to hallucinations.

And if you try JSON prompts, you get consistent outputs:



The reason JSON is so effective is that AI models are trained on massive amounts of structured data from APIs and web applications.

When you speak their "native language," they respond with laser precision!

Let's understand this a bit more.

1) Structure means certainty

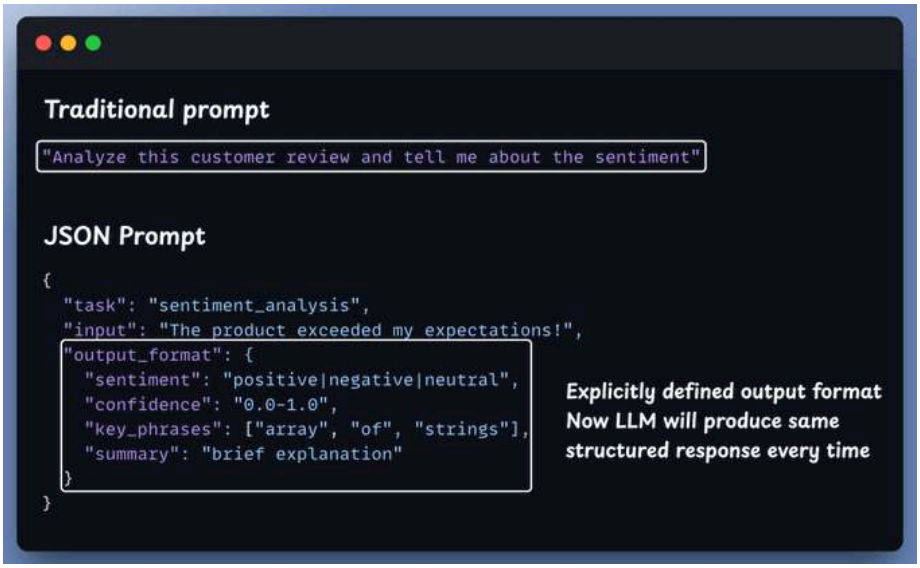
JSON forces you to think in terms of fields and values, which is a gift.

It eliminates gray areas and guesswork.

Here's a simple example:

2) You control the outputs

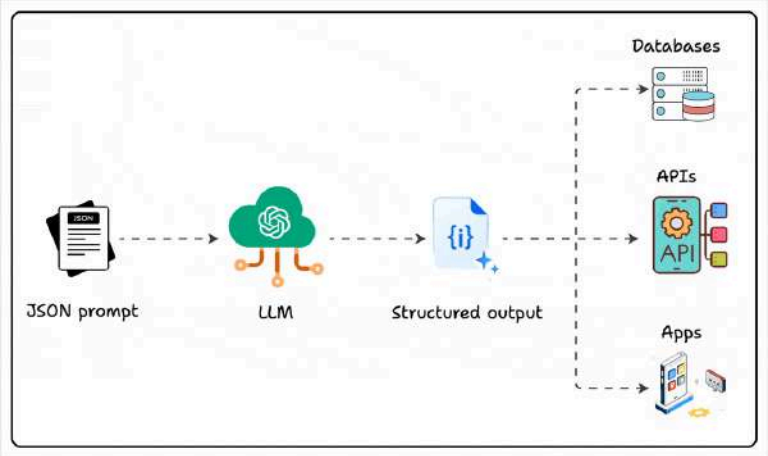
Prompting isn't just about what you ask; it's about what you expect back.



And this works irrespective of what you are doing, like generating content, reports, or insights. JSON prompts ensure a consistent structure every time.

No more surprises, just predictable results!

3) Reusable templates → Scalability, Speed & Clean handoffs



You can turn JSON prompts into shareable templates for consistent outputs.

Teams can plug results directly into APIs, databases, and apps; no manual formatting, so work stays reliable and moves much faster.

So, are json prompts the best option?

Well, good alternatives exist!

Many models excel at other formats:

- Claude handles XML exceptionally well
- Markdown provides structure without overhead

So it's mainly about structure rather than syntax as depicted below:

To summarise:

Structured JSON prompting for LLMs is like writing modular code; it brings clarity of thought, makes adding new requirements effortless, & creates better communication with AI.

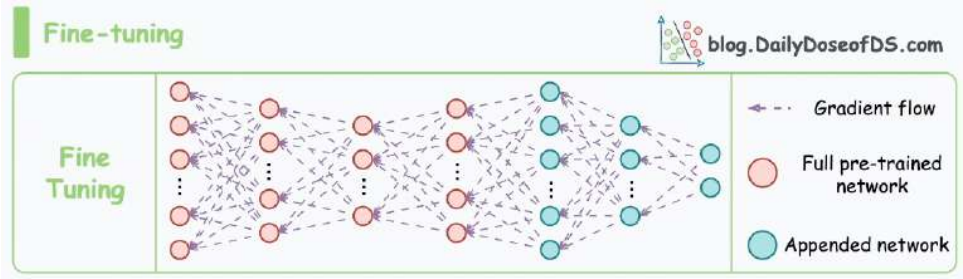
It's not just a technique, but rather evolving towards a habit worth developing for cleaner AI interactions.

Fine-tuning

What is Fine-tuning?

In the pre-LLM era, whenever someone open-sourced any high-utility model for public use, in most cases, practitioners would fine-tune that model to their specific task.

Fine-tuning means adjusting the weights of a pre-trained model on a new dataset for better performance. This is neatly depicted in the diagram below:



When the model was developed, it was trained on a specific dataset that might not perfectly match the characteristics of the data a practitioner wants to use it on.

The original dataset might have had slightly different distributions, patterns, or levels of noise compared to the new dataset.

Fine-tuning allows the model to adapt to these differences, learning from the new data and adjusting its parameters to improve its performance on the specific task at hand.

For instance, consider BERT. It's a Transformer-based language model, which is popularly used for text-to-embedding generation (92k+ citations on the original paper).

It's open-source.

BERT was pre-trained on a large corpus of text data, which might be very very different from what someone else may want to use it on.

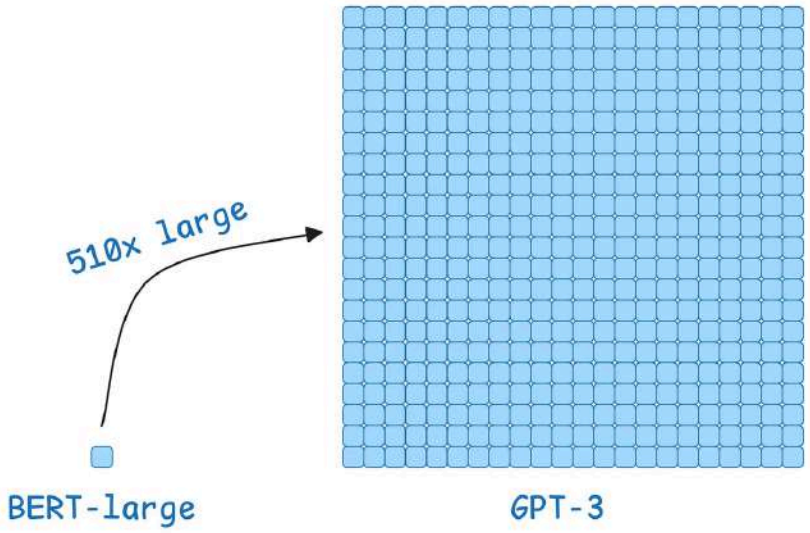
Thus, when using it on any downstream task, we can adjust the weights of the BERT model along with the augmented layers, so that it better aligns with the nuances and specificities of the new dataset.

Issues with traditional fine-tuning

However, a problem arises when we use the same traditional fine-tuning technique on much larger models - LLMs, for instance.

This is because, as you may already know, these models are huge - billions or even trillions of parameters.

Consider the size difference between BERT-large and GPT-3:

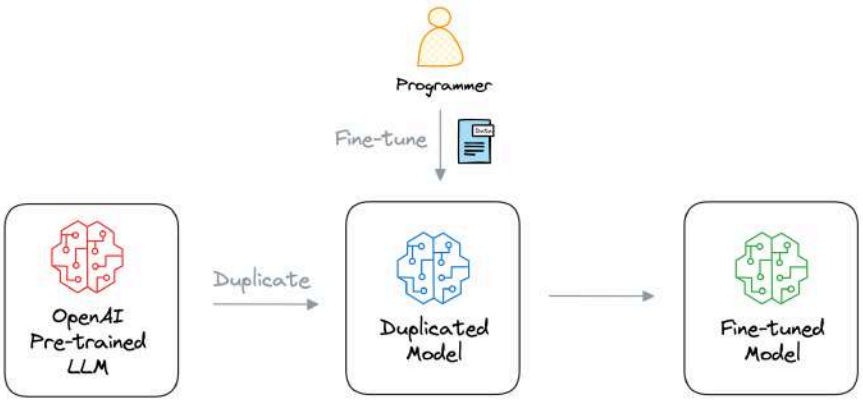


Fine-tuning BERT-large on a single GPU is easy with traditional fine-tuning.

But it's impossible with GPT-3, which has 175B parameters.

That's 350GB of memory just to store model weights (float16 precision).

Imagine OpenAI used traditional fine-tuning within its fine-tuning API:

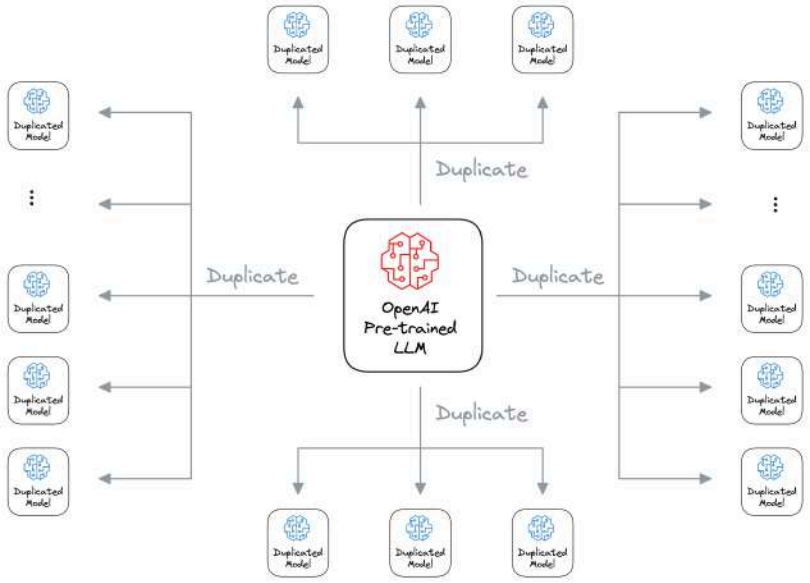


- If 10 users fine-tuned GPT-3 → 3500 GB to store weights.
- If 1000 users fine-tuned GPT-3 → 350k GB to store weights.
- If 100k users fine-tuned GPT-3 → 35M GB to store weights.

And the problems don't end there:

- OpenAI bills solely based on usage.
    - What if someone fine-tunes the model for learning but never uses it?
- Since a request can come anytime, should they always keep the fine-tuned model in memory since loading 350GB is a heavy task?

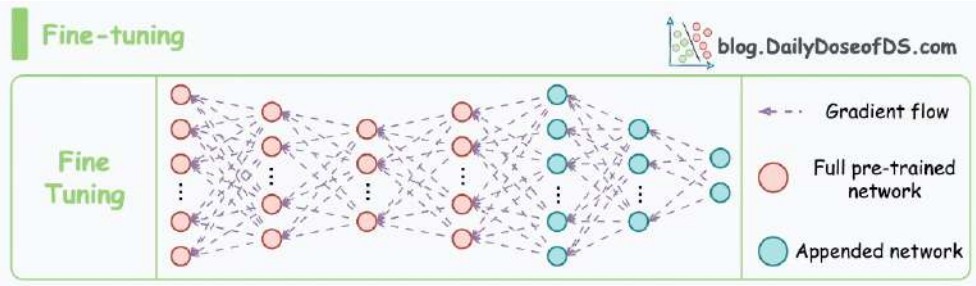
Traditional fine-tuning is just not practically feasible here, and in fact, not everyone can afford to do it due to a lack of massive infrastructure.



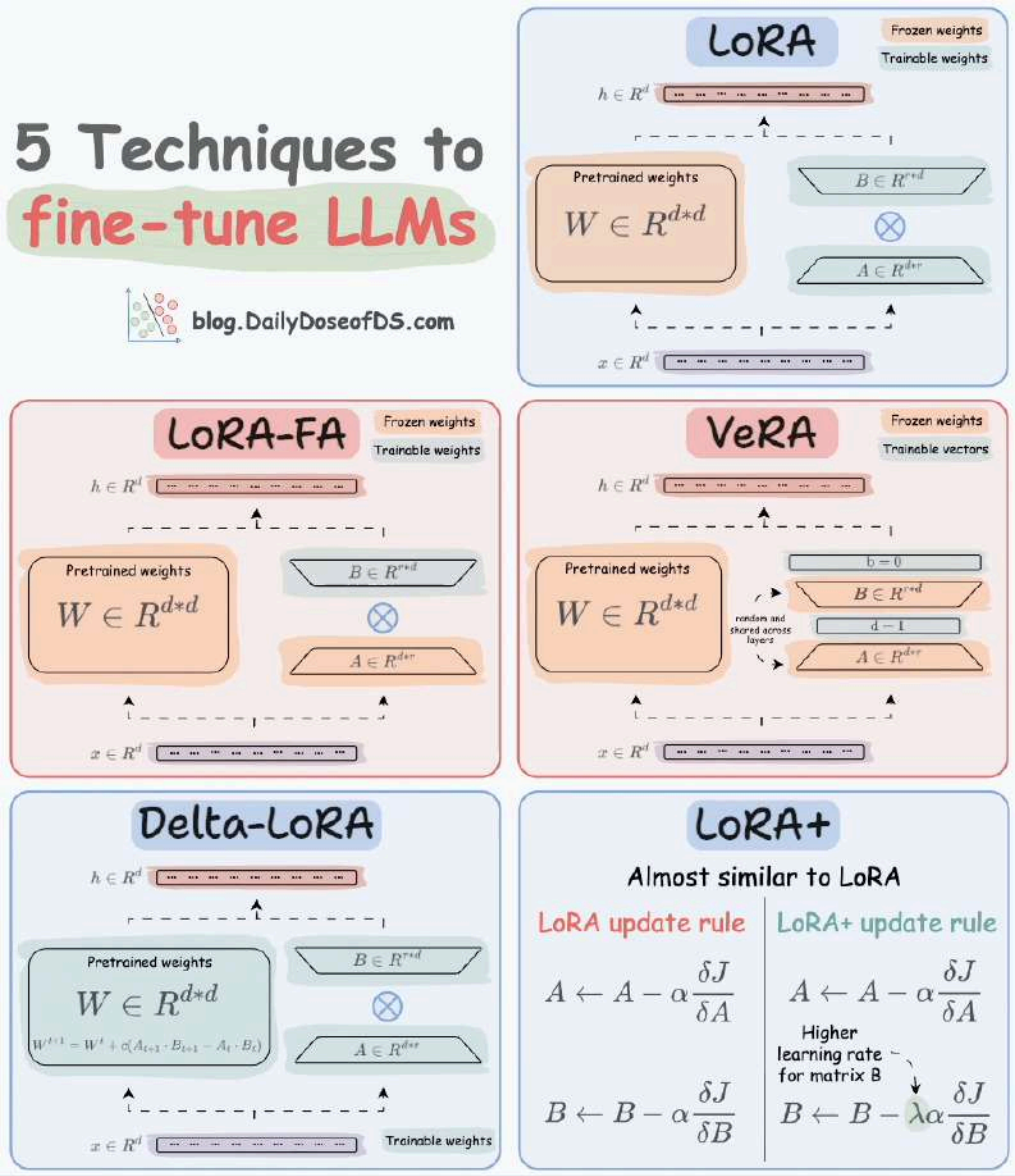
Additionally, maintaining the infrastructure to support fine-tuning requests from potentially thousands of customers simultaneously would be a huge task for them.

5 LLM Fine-tuning Techniques

Traditional fine-tuning (depicted below) is infeasible with LLMs because these models have billions of parameters and are hundreds of GBs in size, and not everyone has access to such computing infrastructure.



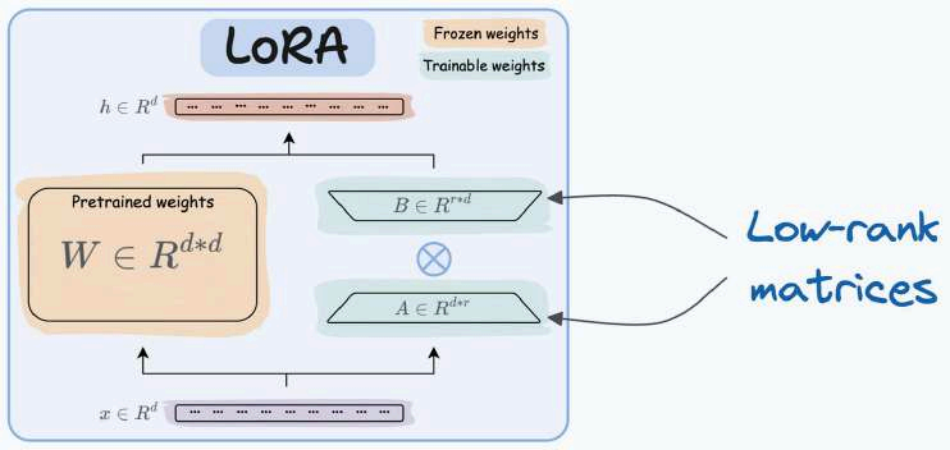
Thankfully, today, we have many optimal ways to fine-tune LLMs, and five such popular techniques are depicted below:



Let's understand these:

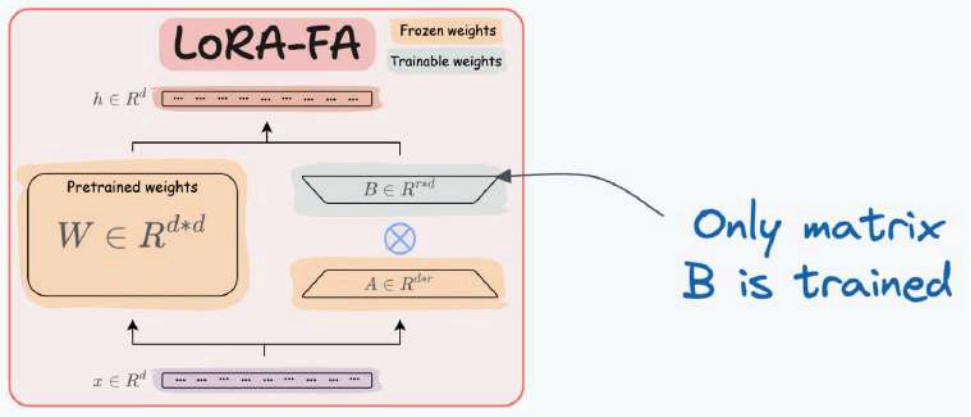
1) LoRA

Add two low-rank matrices A and B alongside weight matrices, which contain the trainable parameters. Instead of fine-tuning W, adjust the updates in these low-rank matrices.



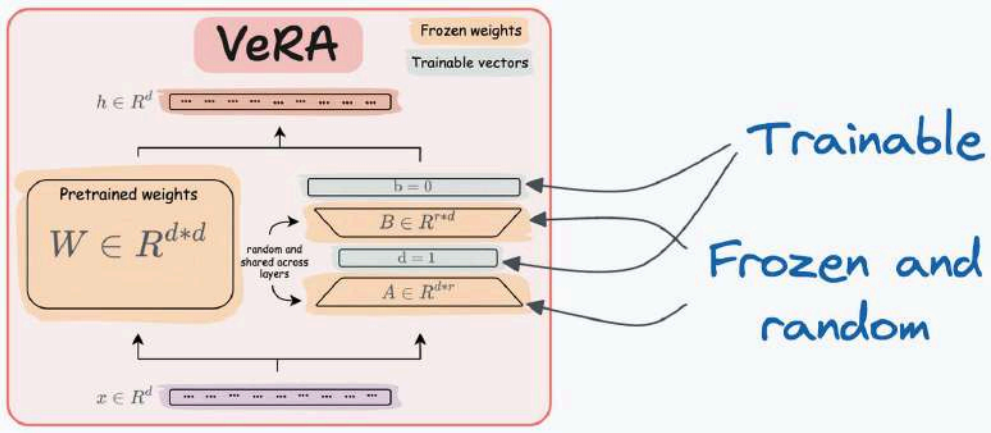
2) LoRA-FA

While LoRA considerably decreases the total trainable parameters, it still requires substantial activation memory to update the low-rank weights. LoRA-FA (FA stands for Frozen-A) freezes the matrix A and only updates matrix B.



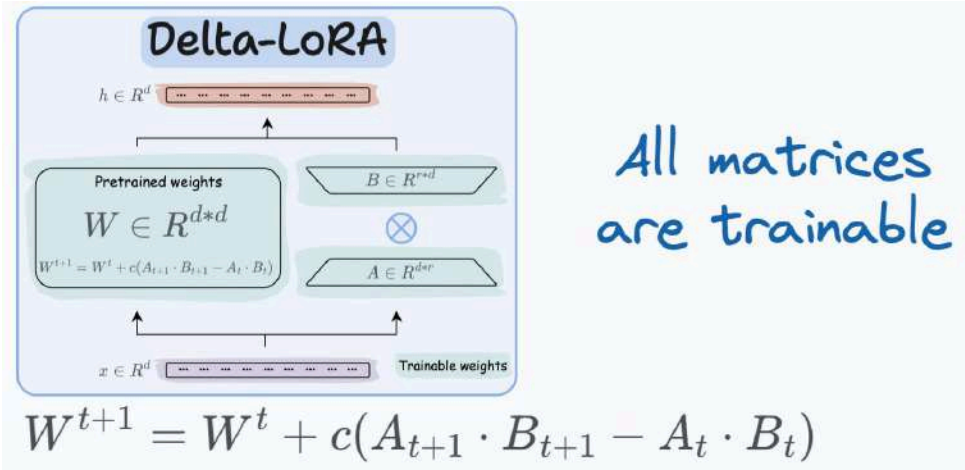
3) VeRA

In LoRA, every layer has a different pair of low-rank matrices A and B, and both matrices are trained. In VeRA, however, matrices A and B are frozen, random, and shared across all model layers. VeRA focuses on learning small, layer-specific scaling vectors, denoted as b and d, which are the only trainable parameters in this setup.



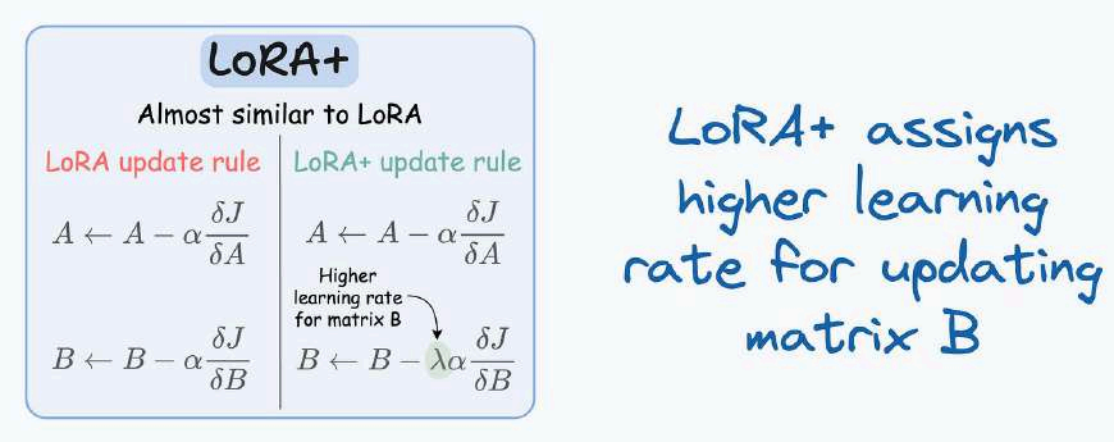
4) Delta-LoRA

Here, in addition to training low-rank matrices, the matrix W is also adjusted but not in the traditional way. Instead, the difference (or delta) between the product of the low-rank matrices A and B in two consecutive training steps is added to W:



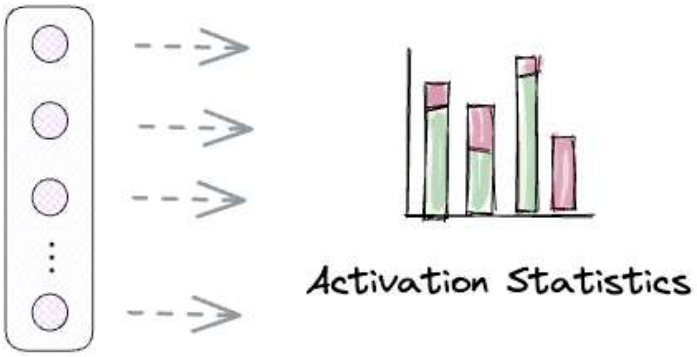
5) LoRA+

In LoRA, both matrices A and B are updated with the same learning rate. Authors found that setting a higher learning rate for matrix B results in more optimal convergence.

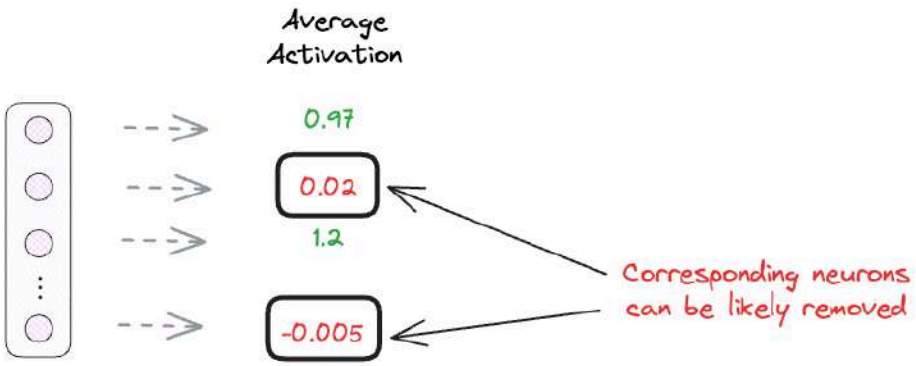


Bonus: LoRA-drop

LoRA-drop observes that not all layers benefit equally from LoRA updates. It first adds low-rank matrices to every layer and trains briefly, then measures each layer's activation strength to see which layers actually matter.



Layers whose LoRA activations stay near zero have minimal influence on the model's output and can be removed.

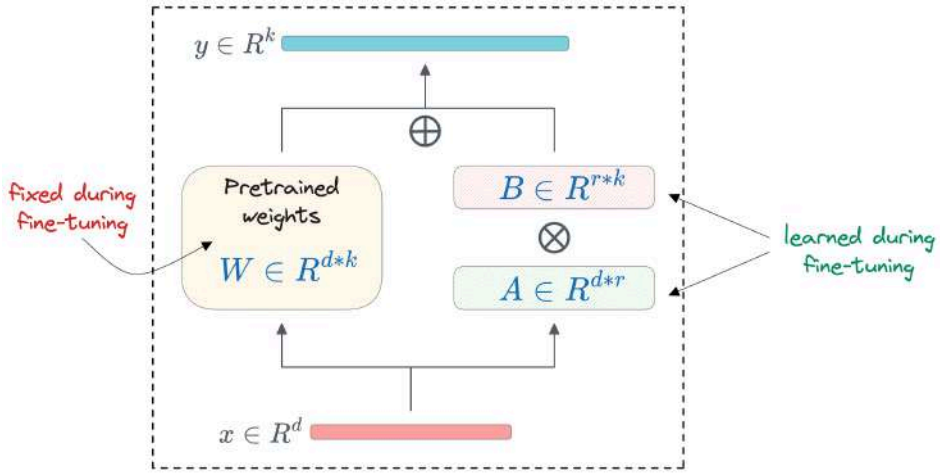


By keeping LoRA only in high-impact layers, LoRA-drop reduces training cost and speeds up fine-tuning with little to no loss in accuracy.

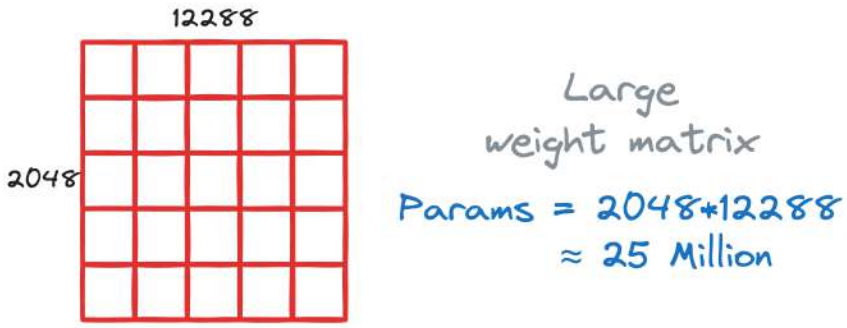
Bonus: Quantized Low-Rank Adaptation (QLoRA)

Quantized Low-Rank Adaptation (QLoRA) is an improvement on the LoRA technique discussed above, which further addresses the memory limitations associated with fine-tuning large models using LoRA.

More specifically, if we recall what we discussed above in LoRA, we saw that we augment the network layers whose weights are W with two matrices A and B.



Now, considering the example where we have 25 Million parameters in the weight matrix W:

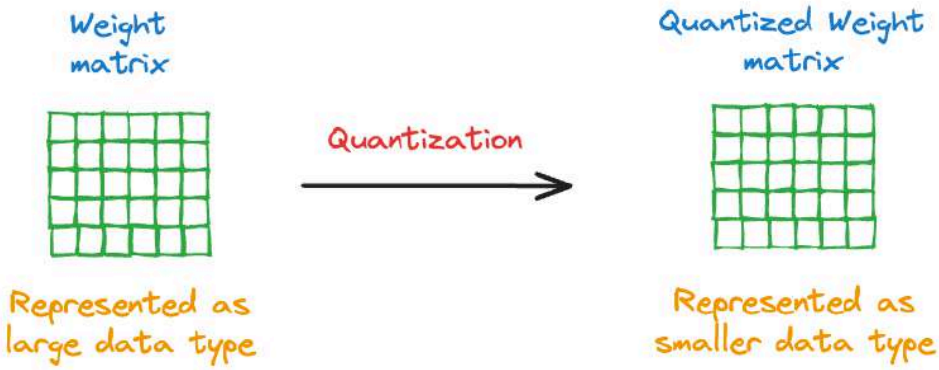


Typically, these 25 million parameters will be represented as float32, which requires 32 bits (or 4 bytes) per parameter. This leads to a significant memory footprint, especially for large LLMs.

This results in a memory utilization of (25 million * 4 bytes/parameter) = 100 million bytes for this matrix alone, which is 0.1GBs.

The idea in QLoRA is to reduce this memory utilization of weight matrix W using Quantization.

As you may have guessed, Quantization involves using lower-bit representations, such as 16-bit, 8-bit, or 4-bit, to represent parameters.



This results in a significant decrease in the amount of memory required to store the model's parameters.

For instance, consider your model has over a million parameters, each represented with 32-bit floating-point numbers.

If possible, representing them with 8-bit numbers can result in a significant decrease (~75%) in memory usage while still allowing for a large range of values to be represented.

Of course, Quantization introduces a trade-off between model size and precision.

While reducing the bit-width of parameters makes the model smaller, it also leads to a loss of precision.

This means the model's predictions become more somewhat approximate than the original, full-precision model.

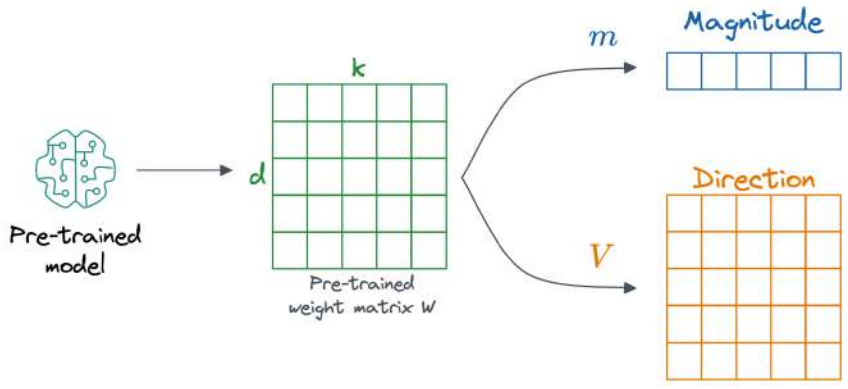
QLoRA does employ some special techniques to preserve the information as much as possible, but there is definitely some trade-off involved.

This somewhat lies along the lines of Quantization in Model compression, which we will cover ahead in the LLM optimization section.

Bonus: DoRA

DoRA (Weight-Decomposed Low-Rank Adaptation) represents a refined approach to fine-tuning large models by addressing a key limitation of LoRA (Low-Rank Adaptation) while preserving its efficiency.

At its core, DoRA builds upon the principles of LoRA but introduces a decomposition step that separates a pretrained weight matrix W into two components: magnitude (m) and direction (V).

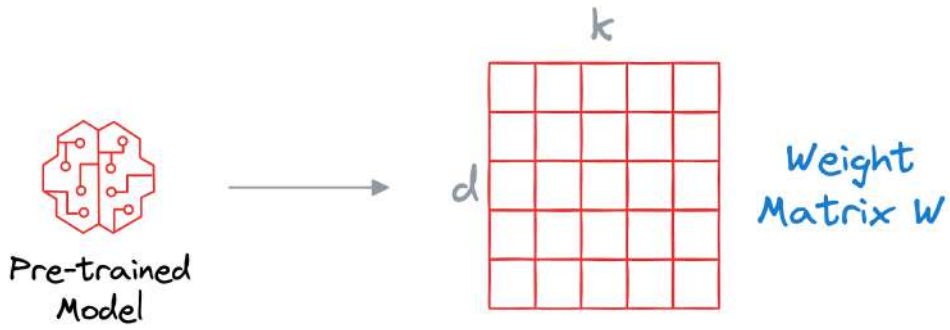


This separation allows the fine-tuning process to target these components independently, improving parameter efficiency and performance.

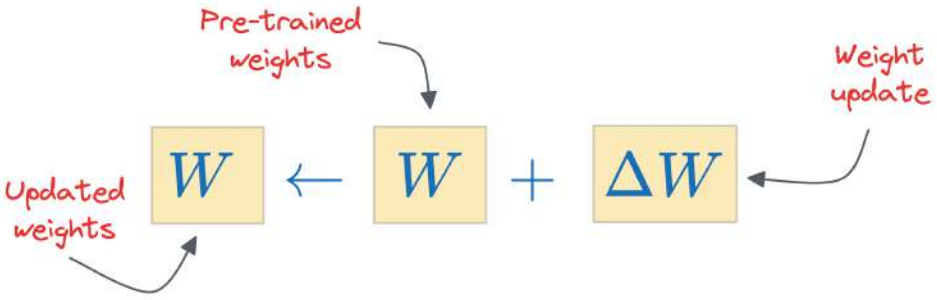
Implementing LoRA From Scratch

Let us understand LoRA in more detail.

Consider the current weights of some random layer in the pre-trained model are W of dimensions d * k, and we wish to fine-tune it on some other dataset.



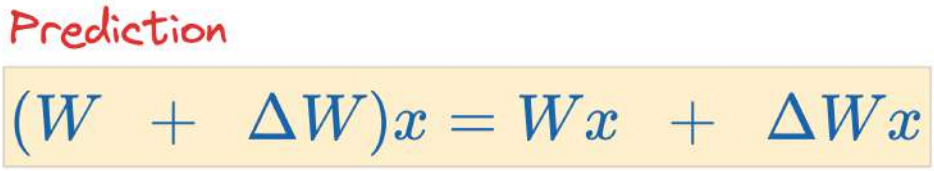
During fine-tuning, the gradient update rule suggests that we must add ΔW to get the updated parameters:



For simplicity, you can think about ΔW as the update obtained after running gradient descent on the new dataset:

Also, instead of updating the original weights W, it is perfectly legal to maintain both matrics, Wand ΔW.

During inference, we can compute the prediction on an input sample x as follows:



In fact, in all the model fine-tuning iterations, W can be kept static, and all weight updates using gradient computation can be incorporated to ΔW instead.

But you might be wondering...how does that even help?

The matrix W is already huge, and we are talking about introducing another matrix that is equally big.

So, we must introduce some smart tricks to manipulate ΔW so that we can fulfill the fine-tuning objective while ensuring we do not consume high memory.

Now, we really can't do much about W as these weights refer to the pre-trained model. So all optimization (if we intend to use any) must be done ΔW instead.

While doing so, we must also remember that currently, both W and ΔW have the same dimensions.But given that W already is huge, we must ensure that ΔW does not end up being of the same dimensions, as this will defeat the entire purpose of efficient fine-tuning.

In other words, if we were to keep ΔW of the same dimensions as W, then it would have been better if we had fine-tuned the original model itself.

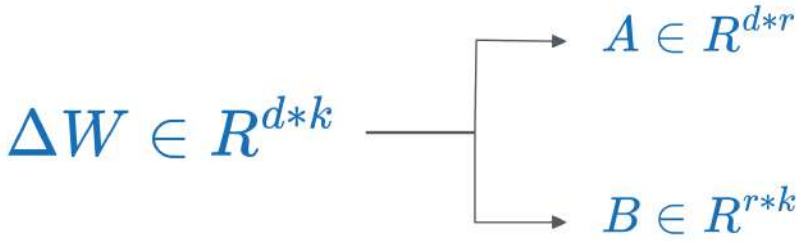
How does LoRA work?

Now, you might be thinking...

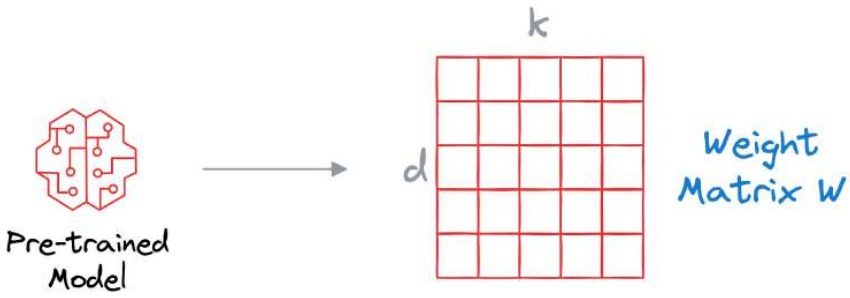
But how can we even add two matrics if both have different dimensions?

It's true, we can't do that.

More specifically, during fine-tuning, the weight matrix W is frozen, so it does not receive any gradient updates. Thus, all gradient updates are redirected to the ΔW matrix. But to ensure that ΔW and W remain additive to generate a final representation for the fine-tuned model, the ΔW matrix is split into a product of two low-rank matrices A and B, which contain the trainable parameters.



As discussed earlier, the dimensions of W are d * k:



Thus, the dimensions of ΔW must also be d * k. But this does not mean that the total trainable parameters in A and B matrix must also align with the dimensions of ΔW.

Instead, A and B can be extremely small matrices, and the only thing we must ensure is that their product results in a matrix, which has dimensions d * k.

Thus:

The dimension of matrix A is set to d * r.

The dimension of matrix B is set to r * k.

If we check their product, that is indeed d * k.

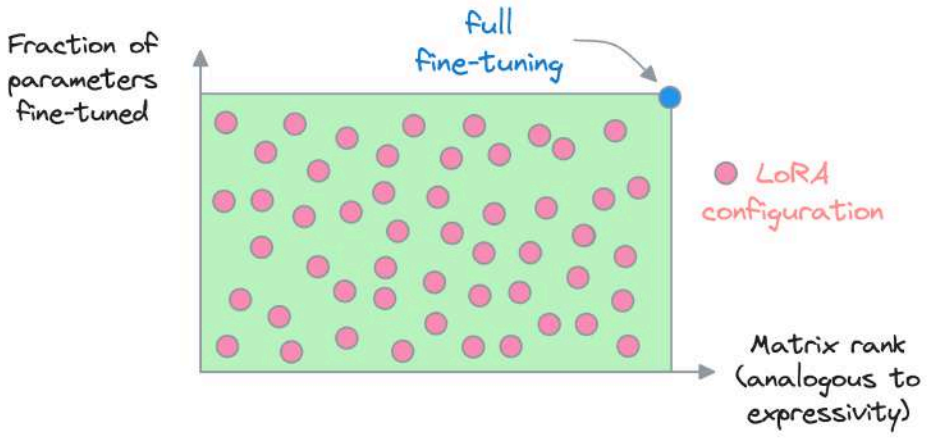
During training, only matrix A and B are trained while the entire network's weights are kept fixed.

This is how LoRA works.

To discuss it more formally, LoRA questions the very idea of full-model fine-tuning by asking two questions:

- Do we really need to fine-tune all the parameters in the original model?
- How expressive are the parameters of the original model (or matrix rank)?

This can be plotted as a 2D grid, as shown below:



In the above image, every point denotes a possible LoRA configuration. Also, the upper right corner refers to full fine-tuning.

Experimentally, it is observed that an ideal configuration is located in the bottom left corner of the above grid, which means that we do not need to train all the model parameters.Now that we understand how LoRA works, let's proceed with implementing LoRA.

Implementation

While a few open-source implementations are already available for LoRA, yet, we shall implement it from scratch using PyTorch so that we get a better idea of the practical details.

As discussed above, a typical LoRA layer comprises two matrices, A and B. These have been implemented in the *LoRAWeights* class below along with the forward pass:

As demonstrated above, the *LoRAWeights* class aims to decompose a matrix of dimensionality d * k into two matrices A and B. Thus, it accepts four parameters:

- *d*: The number of rows in matrix W.
- *k*: The number of columns in matrix W.
- *r*: The rank hyperparameter.
- *alpha*: A scaling parameter that controls the strength of the adaptation.

Also, both *self.A* and *self.B* are learnable parameters of the module, representing the matrices used in the decomposition.

- The matrix A has been initialized from a Gaussian distribution.
    - *Note: If needed, we can also scale this matrix A so that the initial values are not too large.*
- The matrix B is a zero matrix.

As discussed earlier, this ensures that the product of AB is zero as we begin fine-tuning. This initialization also validates the fact that if no fine-tuning has been done so far, the original model weights are retained:

$$W + A\overset{0}{\cancel{B}} = W$$

In the *forward* method, the input *x* is multiplied by the matrices *A* and *B*, and then scaled by *alpha*. The result is returned as the output of the module.

The parameter *alpha* is another hyperparameter, which acts as a scaling factor. It determines the impact of the new layers on the current model.

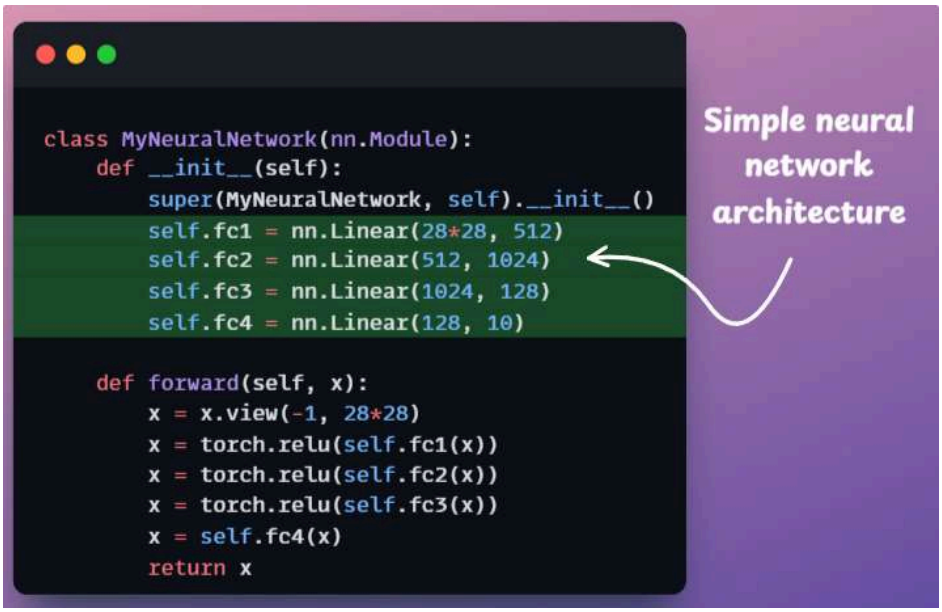
Prediction

$$(W + \alpha\Delta W)x = Wx + \alpha ABx$$

- A higher value of *alpha* means that the changes made by the LoRA layer will be more significant, potentially leading to more pronounced adjustments in the model's behavior.

- Conversely, a lower value of *alpha* results in more subtle changes, as the impact of the transformation is reduced.

As discussed earlier, LoRA is used on large matrices of a neural network. For instance, say we have the following neural network class:



As LoRA is used after training, so we will already have a trained model available. Let's say it is accessible using the *model* object.

Now, our primary objective is to attach the matrices in *LoRAWeights* class with the matrices in the layers of the above network. And, of course, each layer (*fc1*, *fc2*, *fc3*, *fc4*) will have its respective *LoRAWeights* layer.

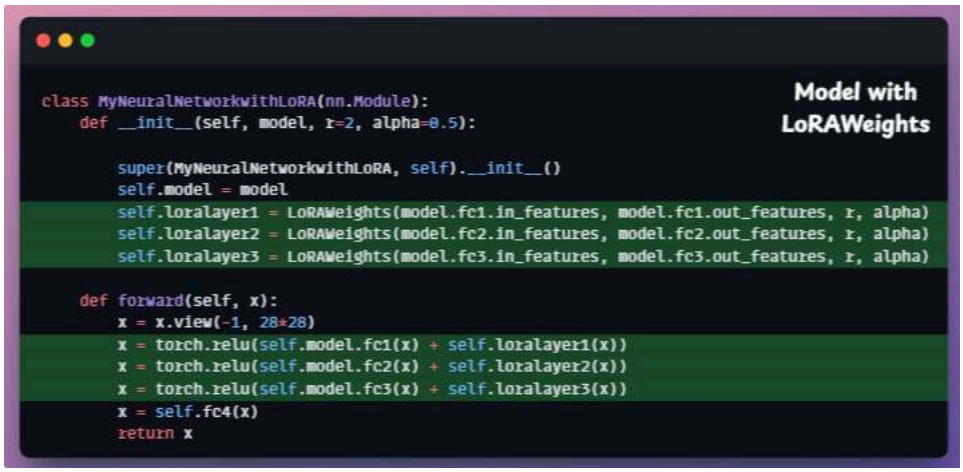
Note: Of course, it is not necessary that each layer must have a respective fine-tuning LoRAWeights layer too. In fact, in the original paper, it is mentioned that they limited the study to only adapting the attention weights for downstream tasks and they froze the multi-layer perception (feedforward) units of the Transformer for parameter efficiency.

In our case, for instance, we can freeze the *fc4* layer as it is not enormously big compared to other layers in the network.

Also, we must remember that the network is trained as we would usually train any other neural network, but while only training the weight matrices A and B, i.e., the pre-trained model (model) is frozen. We do this as follows:

Done!

Next, we utilize the *LoRAWeights* class to define the fine-tuning network below:



As depicted above:

- The LoRA layers are applied over the fully connected layer (*fc1*, *fc2*, *fc3*) in the existing model. More specifically, we create three *LoRAWeights* layers (*loralayer1*, *loralayer2*, *loralayer3)* based on the dimensions of the fully connected layers (*fc1*, *fc2*, *fc3*) in the *model*.
- In the *forward* method, we pass the input through the first fully connected layer (*fc1*) of the original model and add the output to the result of the LoRA layer applied to the same input (*self.loralayer1(x)*). Next, we apply a ReLU activation function to the sum. We repeat the process for the second

and third fully connected layers (*fc2*, *fc3*) and lastly, return the final output of the last fully connected layer of the pre-trained model (*fc4*).

Done!

Now this *MyNeuralNetworkwithLoRA* model can be trained like any other neural network.

We have ensured that the pre-trained model (*model*) does not update during fine-tuning and only weights in *LoRAWeights* class is learned.

So far, we explored how to update model weights efficiently (LoRA and its variants).

But fine-tuning also depends on what data you use to update the model.

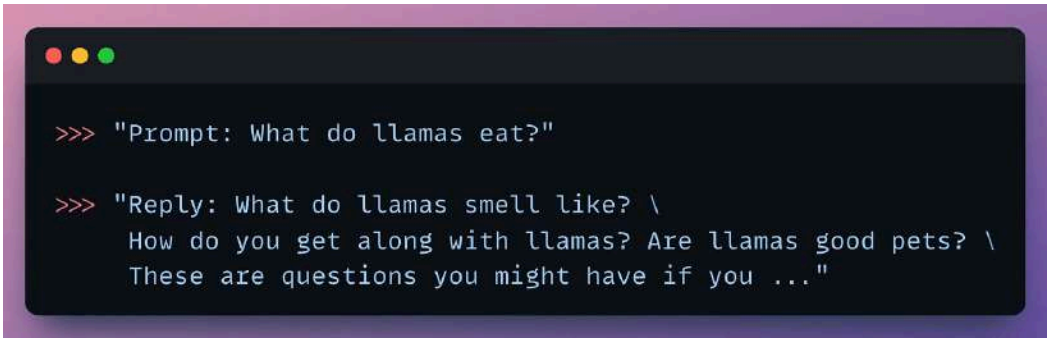
This brings us to instruction fine-tuning (IFT) - the process of teaching an LLM how to follow human instructions by training it on curated instruction–response pairs.

IFT is the foundation of supervised fine-tuning (SFT), and most modern LLMs rely on some form of IFT during alignment.That was pretty simple, wasn't it?

Generate Your Own LLM Fine-tuning Dataset(IFT)

Once an LLM has been pre-trained, it simply continues the sentence as if it is one long text in a book or an article.

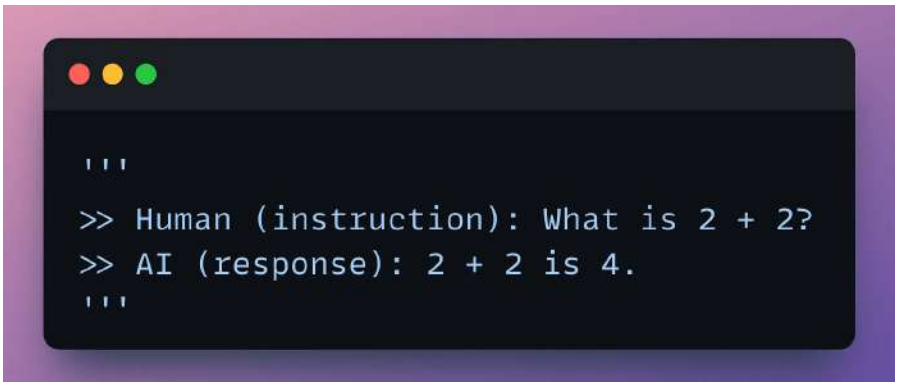
For instance, check this to understand how a pre-trained LLM behaves when prompted:



Generating a synthetic dataset using existing LLMs and utilizing it for fine-tuning can improve this.

The synthetic data will have fabricated examples of human-AI interactions.

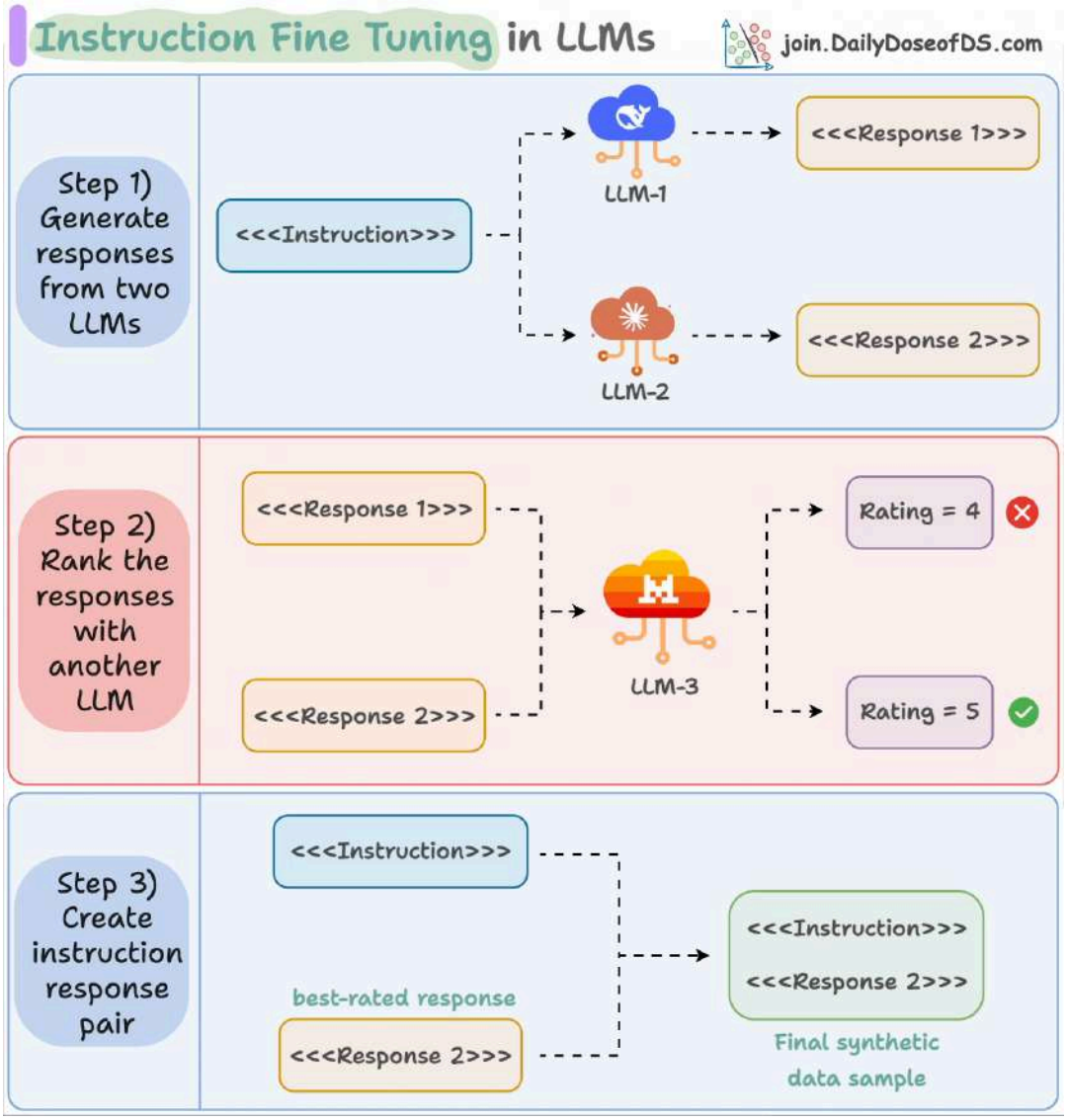
Check this sample:



This process is called instruction fine-tuning and it is described in the animation below:

Distilabel is an open-source framework that facilitates generating domain-specific synthetic text data using LLMs.

Check this to understand the underlying process:



- Input an instruction.
- Two LLMs generate responses.
- A judge LLM rates the responses.
- The best response is paired with the instruction.

And you get the synthetic dataset!

Next, let's look at the code.

First, we start with some standard imports:

synthetic_data.py

```python
import pandas as pd

from distilabel.llms import OllamaLLM
from distilabel.pipeline import Pipeline
from distilabel.steps import LoadDataFromHub
from distilabel.steps.tasks import TextGeneration, UltraFeedback
from distilabel.steps import GroupColumns
```

Next, we load the Llama-3 models locally with Ollama:

synthetic_data.py

```python
model1 = OllamaLLM(model="llama3.1", timeout=1000)

model2 = OllamaLLM(model="llama3.1:70b-instruct-q2_k", timeout=1000)
```

Moving on, we define our pipeline:

synthetic_data.py

```python
with Pipeline(name="preference-datagen-llama3") as pipeline:

    # load dataset with prompts
    load_dataset = LoadDataFromHub(
        name="load_dataset",
        output_mappings={"prompt": "instruction"},
    )

    # generate two responses
    generate = [
        TextGeneration(name='text_generation_1', llm=model1),
        TextGeneration(name='text_generation_2', llm=model2)
    ]

    # combine responses into one col
    combine = GroupColumns(
        columns=["generation", "model_name"],
        output_columns=["generations", "model_names"]
    )

    # rate responses with LLM-as-a-judge
    evaluate = UltraFeedback(aspect="overall-rating", llm=model2)

    # define and run pipeline
    load_dataset >> generate >> combine >> evaluate
```

- First, we load the dataset (we'll pass it shortly).
- Next, we generate two responses.
- Once done, we combine the responses into one column (under the hood, a prompt template is also created for the third LLM).
- Moving on, we evaluate the responses with an LLM.
- Finally, we define and run the pipeline.

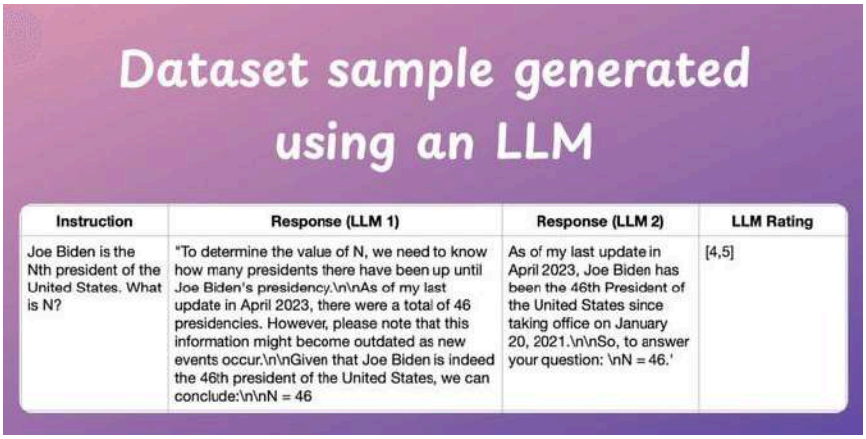
Once the pipeline has been defined, we need to execute it by giving it a seed dataset.

The seed dataset helps it generate new but similar samples. So we execute the pipeline with our seed dataset as follows:

Done!

This produces the instruction and response synthetic dataset as desired.

Check the sample below:



That was simple, wasn't it?

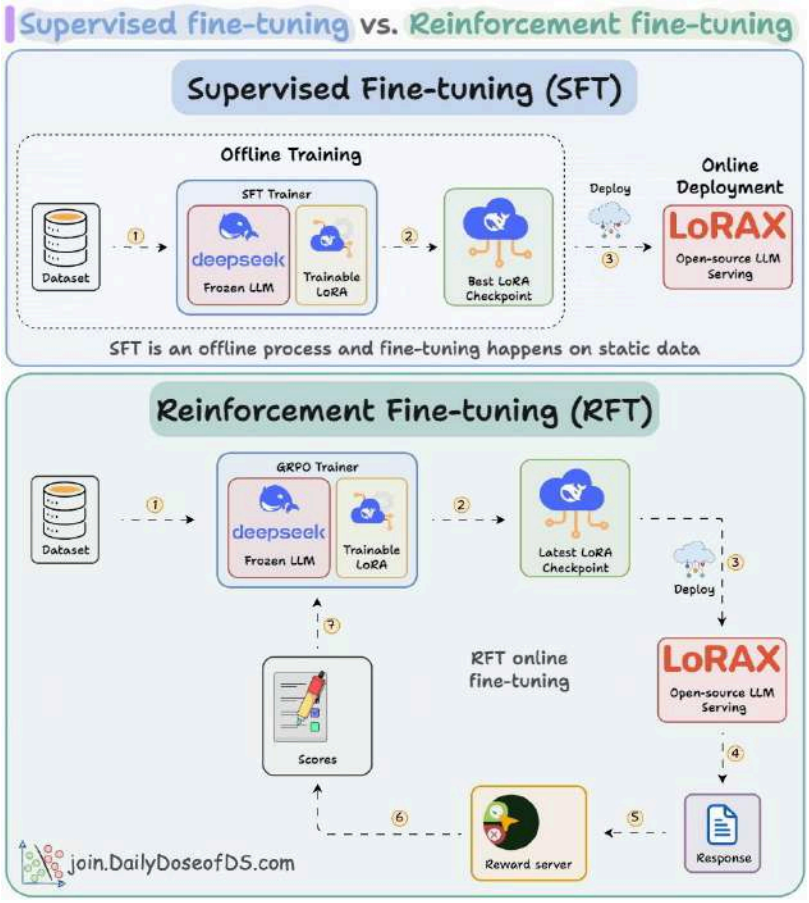
This produces a dataset on which the LLM can be easily fine-tuned.

So far, we've explored how to fine-tune a model efficiently using LoRA and its variants, and what kind of data is typically used through instruction fine-tuning.

The next question is: how do different fine-tuning objectives actually change the learning process?

Broadly, fine-tuning falls into two categories.

- Supervised Fine-Tuning (SFT)
- Reinforcement Fine-Tuning (RFT)

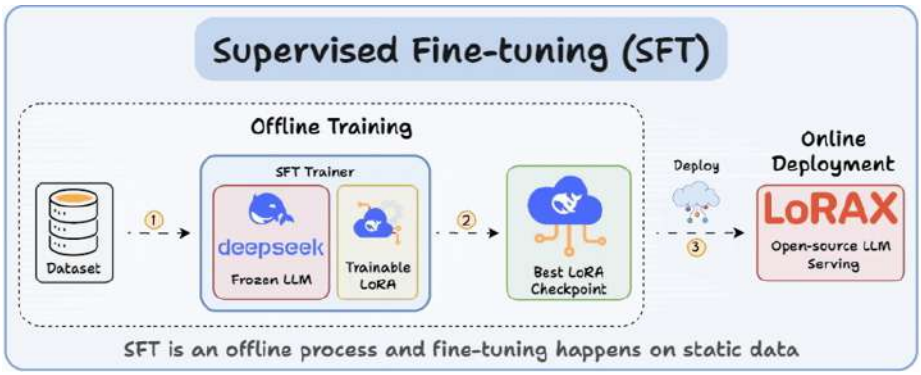


Both update the model using LoRA or similar PEFT methods, but their goals and training signals differ dramatically.

SFT vs RFT

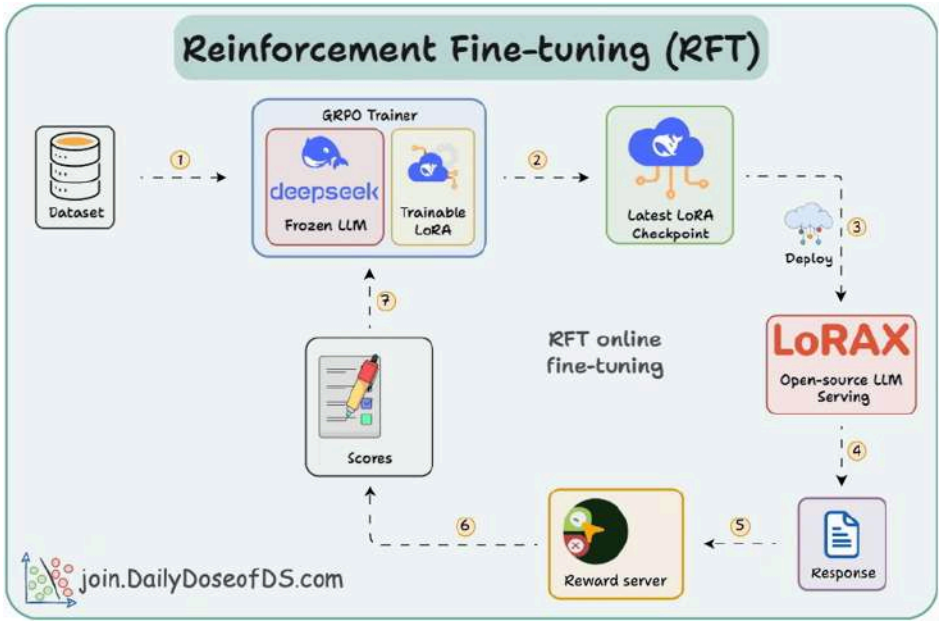
Before diving deeper, it's crucial to understand how we usually fine-tune LLMs using SFT, or supervised fine-tuning.

SFT process:



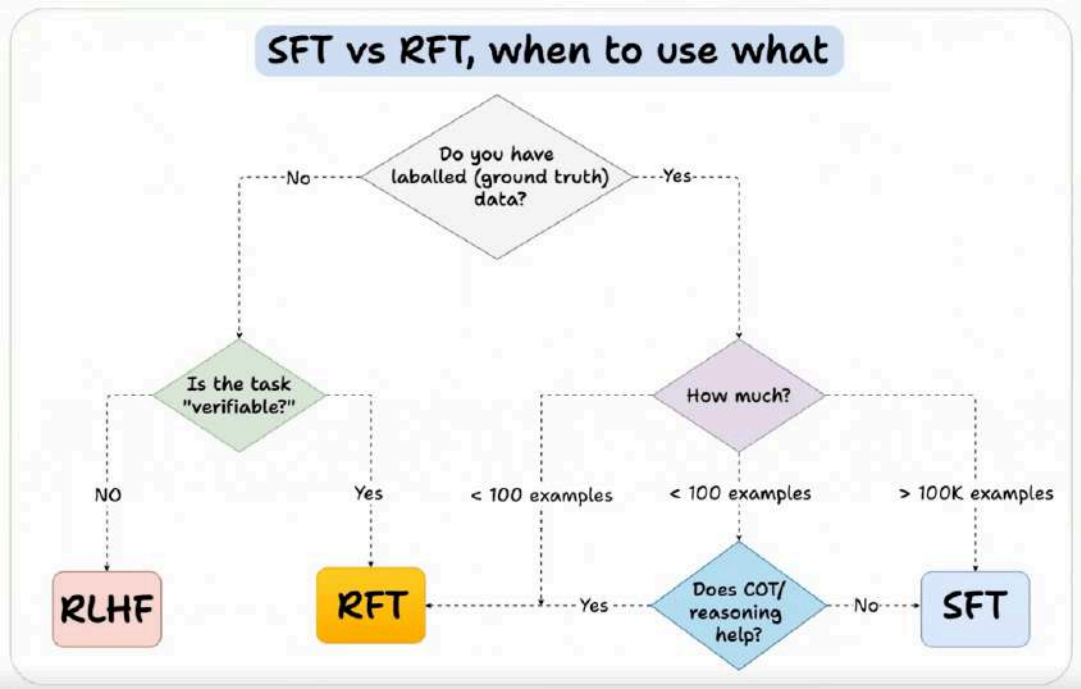
- It starts with a static labeled dataset of prompt–completion pairs.
- Adjust the model weights to match these completions.
- The best model (LoRA checkpoint) is then deployed for inference.

RFT process:



- RFT uses an online "reward" approach - no static labels required.
- The model explores different outputs, and a Reward Function scores their correctness.
- Over time, the model learns to generate higher-reward answers using GRPO.

SFT uses static data and often memorizes answers. RFT, being online, learns from rewards and explores new strategies.



This flowchart gives a quick guide on which fine-tuning method to use based on your data and the nature of the task.

- Start by checking whether you have labelled (ground-truth) data.
- If you don't, the next question is whether the task is verifiable.
    - If not verifiable, you use RLHF, since humans must provide preference signals.
    - If verifiable, RFT works because correctness can be automatically checked.
- If you do have labelled data, the choice depends on how much you have:

- Large datasets → use SFT.
- Tiny datasets → ask if reasoning (like CoT) helps.
  - If yes → RFT
  - If no → SFT

Overall, this decision tree helps you quickly identify the most efficient and reliable fine-tuning strategy for your use case.

Now that we understand when to use SFT or RFT, let's apply RFT in practice.

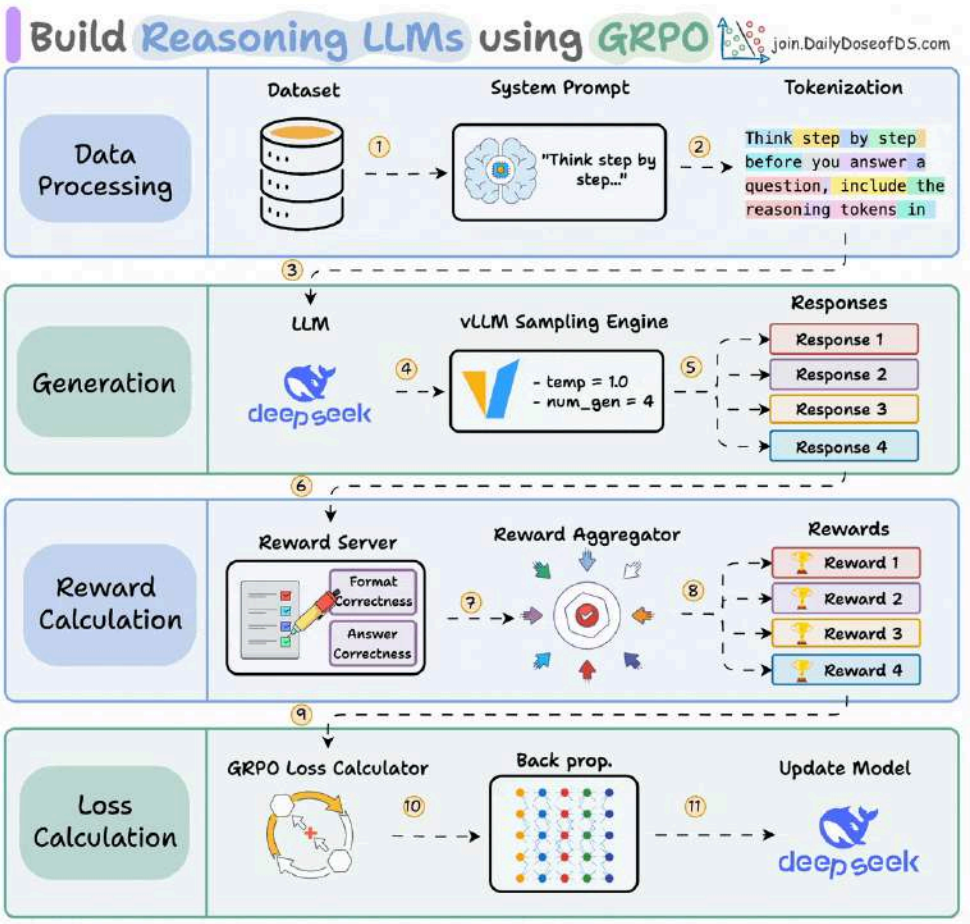
For reasoning-heavy tasks like math or logic, GRPO (Group Relative Policy Optimization) is one of the most effective RFT methods available.

Let's walk through how to fine-tune a model using GRPO with Unsloth.

Build a Reasoning LLM using GRPO [Hands On]

Group Relative Policy Optimization is a reinforcement learning method that fine-tunes LLMs for math and reasoning tasks using deterministic reward functions, eliminating the need for labeled data.

Here's a brief overview of GRPO:



- Start with a dataset and add a reasoning-focused system prompt (e.g., "Think step by step…").
- The LLM generates multiple candidate responses using a sampling engine.
- Each response is assigned rewards, which are aggregated to produce a score for every generated response.
- A GRPO loss function uses these rewards to calculate gradients, backpropagation updates the LLM, and the model improves its reasoning ability over time.

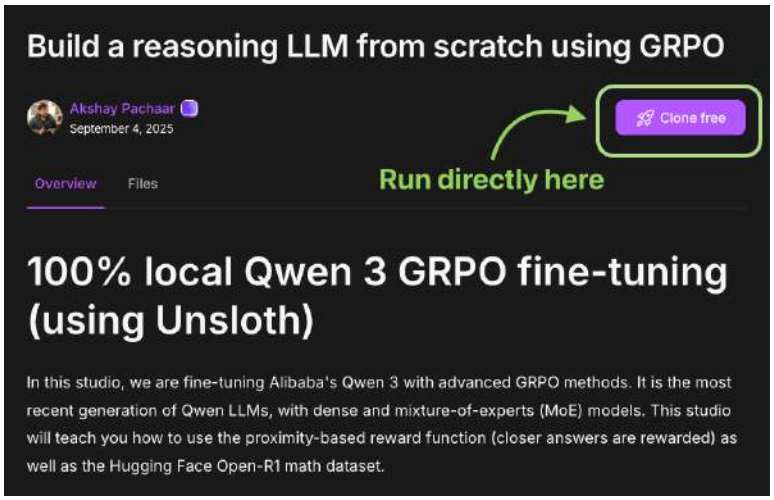
Let's dive into the code to see how we can use GRPO to turn any model into a reasoning powerhouse without any labeled data or human intervention.

We'll use:

- UnslothAI for efficient fine-tuning.

- HuggingFace TRL to apply GRPO.

The code is available here: Build a reasoning LLM from scratch using GRPO. You can run it without any installations by reproducing our environment below:

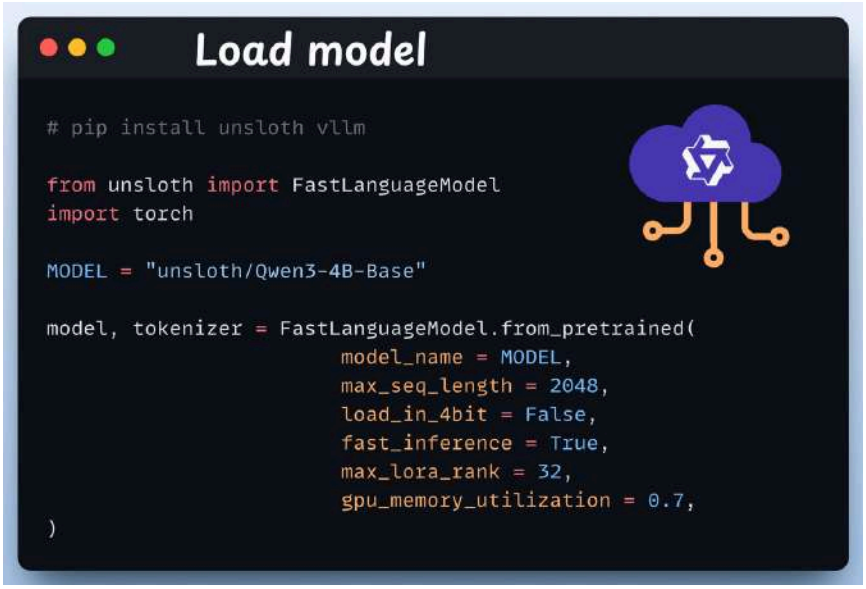


Let's begin!

#1) Load the model

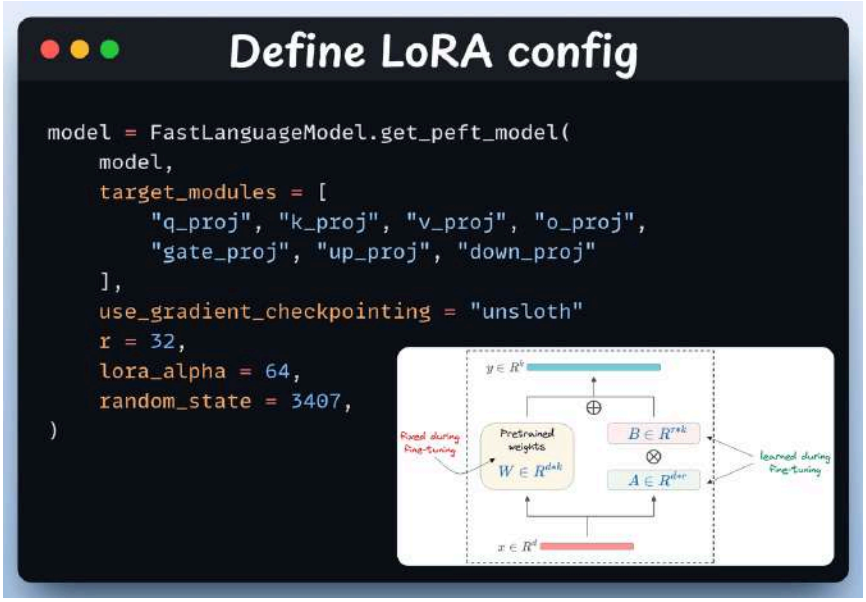
We start by loading Qwen3-4B-Base and its tokenizer using Unsloth.

You can use any other open-weight LLM here.



#2) Define LoRA config

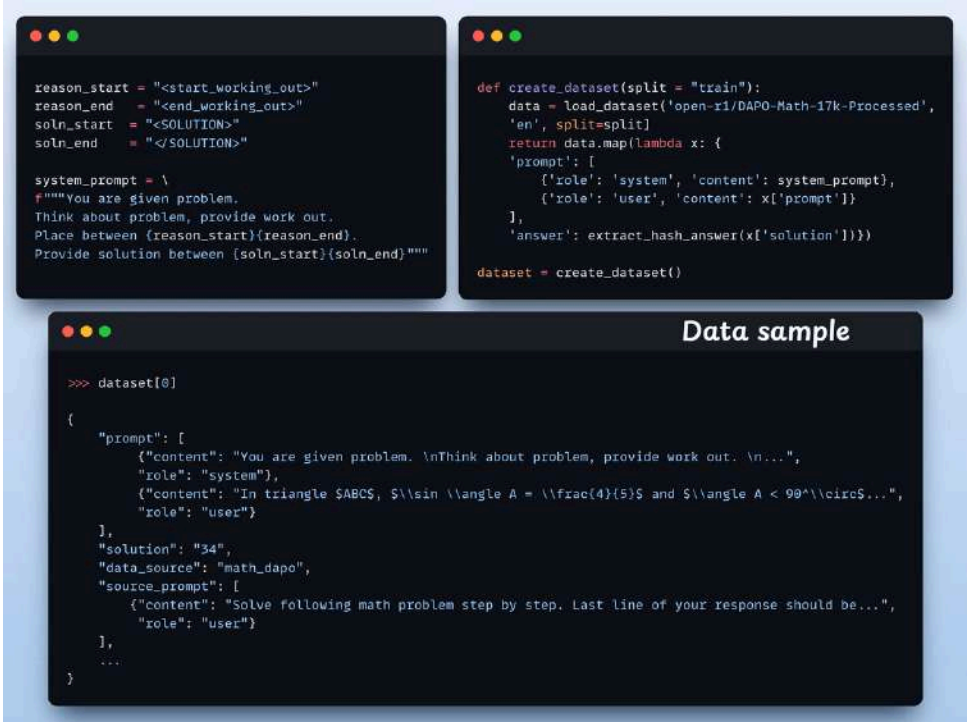
We'll use LoRA to avoid fine-tuning the entire model weights. In this code, we use Unsloth's PEFT by specifying:



- The model
- LoRA low-rank (r)
- Modules for fine-tuning, etc.

#3) Create the dataset

We load the Open R1 Math dataset (a math problem dataset) and format it for reasoning.



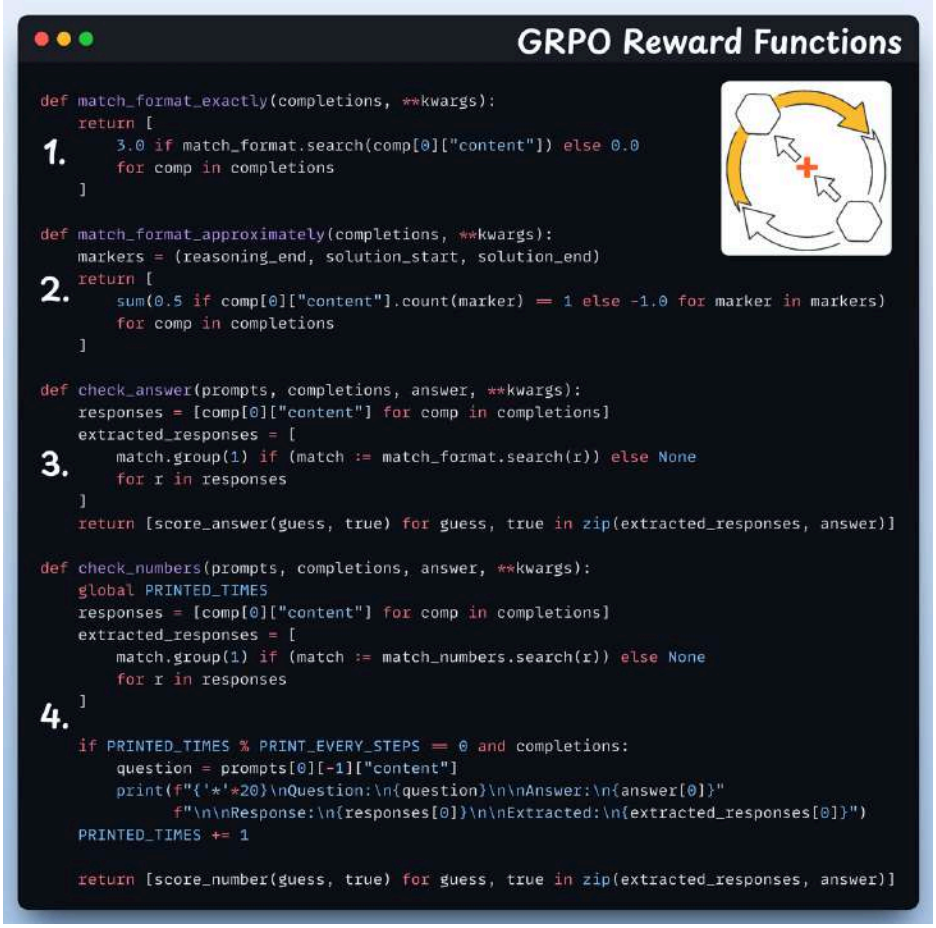
Each sample includes:

- A system prompt enforcing structured reasoning
- A question from the dataset
- The answer in the required format

#4) Define reward functions

In GRPO, we use deterministic functions to validate the response and assign a reward. No manual labelling required!

The reward functions:



- Match format exactly
- Match format approximately
- Check the answer
- Check numbers

#5) Use GRPO and start training

Now that we have the dataset and reward functions ready, it's time to apply GRPO.

HuggingFace TRL provides everything we described in the GRPO diagram, out of the box, in the form of the GRPOConfig and GRPOTrainer.

Comparison

Again, we can see how GRPO turned a base model into a reasoning powerhouse.

RFT methods like GRPO work best when paired with reliable reinforcement learning environments. This brings us to an important component of RL-based fine-tuning: how agents interact with environments.

Bottleneck in Reinforcement Learning

A central difficulty in reinforcement learning lies not in training the agent but in managing the environment in which the agent operates.

The environment defines the task, the rules, the available actions and the reward structure. Because there is no standard way to construct these environments, each project tends to develop its own APIs and interaction patterns.

This fragmentation makes environments difficult to reuse and agents difficult to transfer across tasks. The result is substantial engineering overhead: researchers often spend more time maintaining or re-implementing environments than focusing on learning algorithms or agent behavior.

The Solution: The OpenEnv Framework

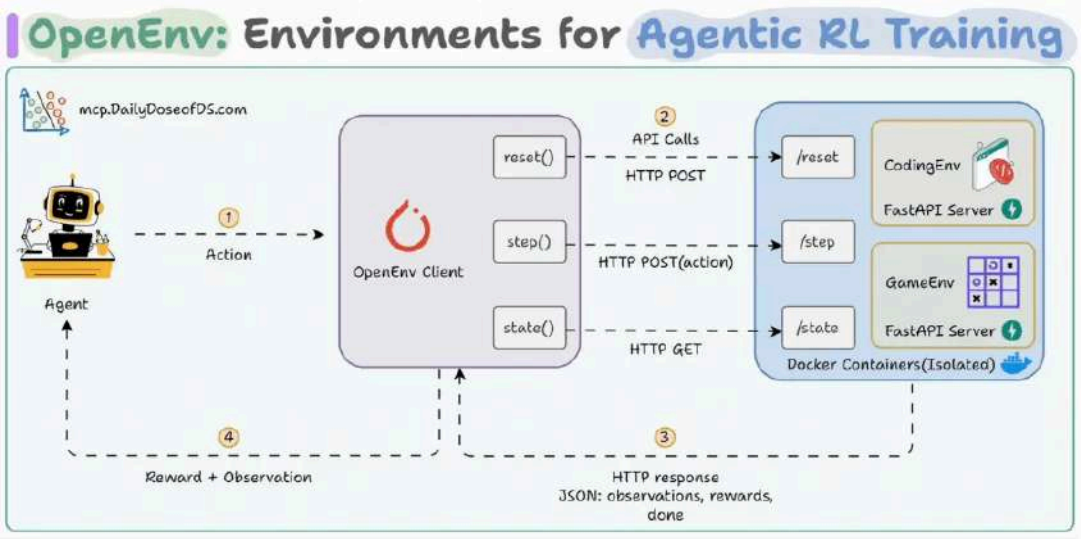
PyTorch OpenEnv is designed to address this lack of standardization. The framework provides a common interface for reinforcement learning environments, inspired by Gymnasium but implemented as a containerized, service-based system.

Each environment exposes three core methods:

- reset() - initialize a new episode
- step(action) - apply an action and receive feedback
- state() - retrieve the current state

Environments run in isolated Docker containers and communicate over HTTP, allowing them to be reproduced, shared, and executed consistently across machines.

The typical workflow proceeds as follows:



- An agent interacts with the environment through an OpenEnv client.
- The client forwards actions to a FastAPI application running inside a Docker container.
- The environment updates its internal state and returns the resulting observations, rewards, and termination status.
- The agent uses this feedback to update its policy and continues the loop.

Because the interface is stable and uniform, the same pattern applies to a wide variety of tasks, from simple games to complex, custom-built worlds.

For a practical demonstration refer Building Agentic RL environments with OpenEnv and Unsloth which demonstrates how to fine-tune the GPT-OSS 20B model with Unsloth to play the game 2048 using the OpenEnv framework.

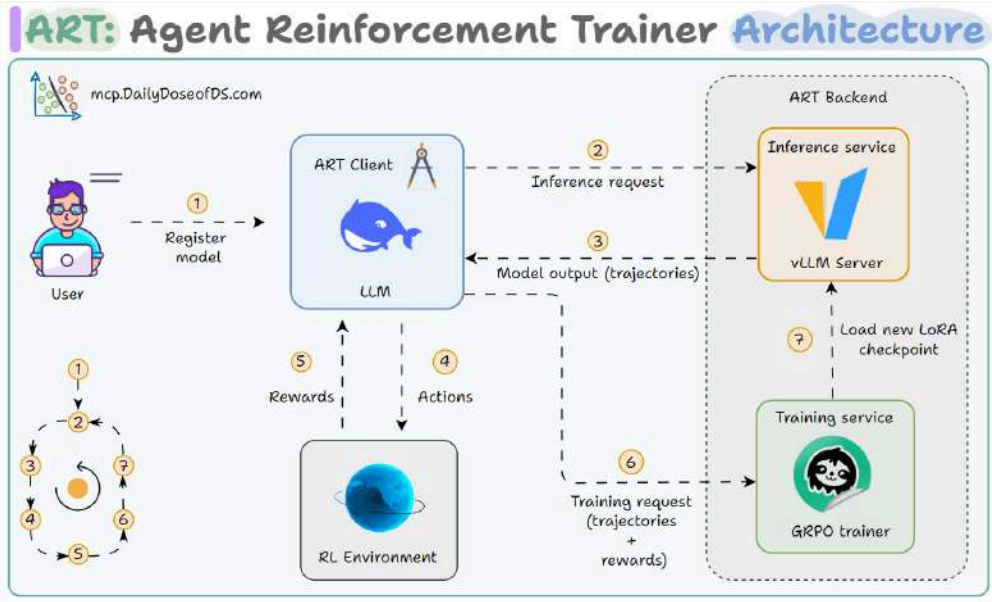
Agent Reinforcement Trainer(ART)

Reinforcement learning becomes more complex when the "agent" is an LLM.

Instead of choosing a simple action like moving left or right - an LLM agent produces multi-step reasoning traces, tool calls, conversations and plans.

Training such agents requires a system that can collect these trajectories, assign rewards and update the model reliably.

ART (Agent Reinforcement Trainer), built by OpenPipe, provides that system.



It is an open-source framework designed specifically for training agentic LLMs from experience. ART handles the pieces that are difficult to engineer manually:

- running the agent to generate full trajectories
- capturing decisions, tool use and reasoning steps
- scoring each trajectory with a custom reward function
- updating the model using reinforcement learning

ART uses a lightweight client that wraps your existing agent with minimal changes. The client communicates with an ART training server, which manages rollouts, reward computation, batching and optimization.

A key feature is ART's support for Group Relative Policy Optimization (GRPO), an RL algorithm widely used for training LLMs. GRPO allows the model to learn from trajectory-level rewards rather than token-level labels, which is essential for improving behaviors like planning, correction and tool use.

The workflow looks like this:

- You start with your existing agent code - ART simply wraps it so you don't need to rewrite anything.
- The agent runs and produces a trajectory.
- The trajectory is scored using a reward function.
- ART applies GRPO (or another supported RL method) to update the policy.
- The loop repeats, gradually improving the agent's behavior.

By handling rollout execution, reward processing and policy optimization, ART lets developers focus on designing effective reward signals and agent strategies rather than building RL infrastructure.

RAG

What is RAG?

Up to this point, we have seen two ways to adapt an LLM to a task:

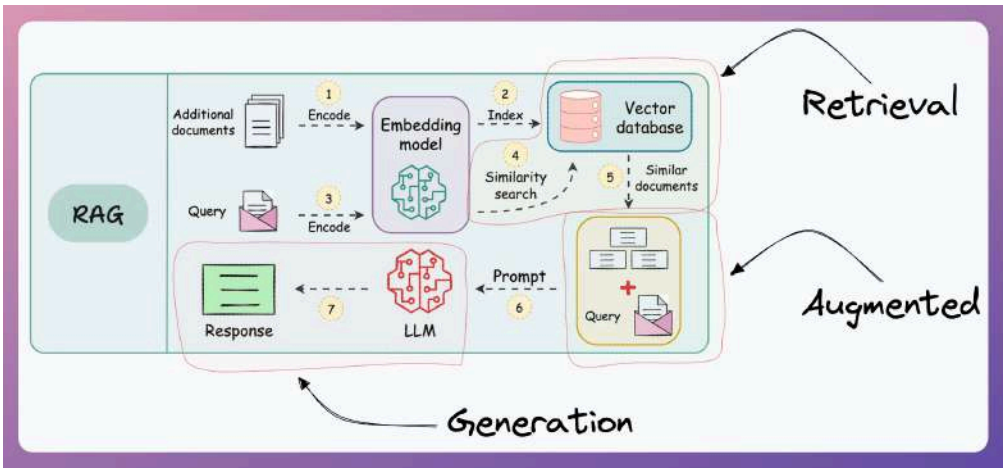
- Prompt Engineering - which steers the model at inference time
- Fine-tuning - which adjusts its internal parameters.

Both approaches are powerful, but they share one fundamental limitation: the model can only use the knowledge it already contains.

LLMs do not automatically know new information, private data, company documents, or anything that appeared after their training cutoff.

Retraining them repeatedly to stay updated is impractical and expensive.

This is where Retrieval-Augmented Generation (RAG) comes in. Let's break it down:



- Retrieval: Accessing and retrieving information from a knowledge source, such as a database or memory.
- Augmented: Enhancing or enriching something, in this case, the text generation process, with additional information or context.
- Generation: The process of creating or producing something, in this context, generating text or language.

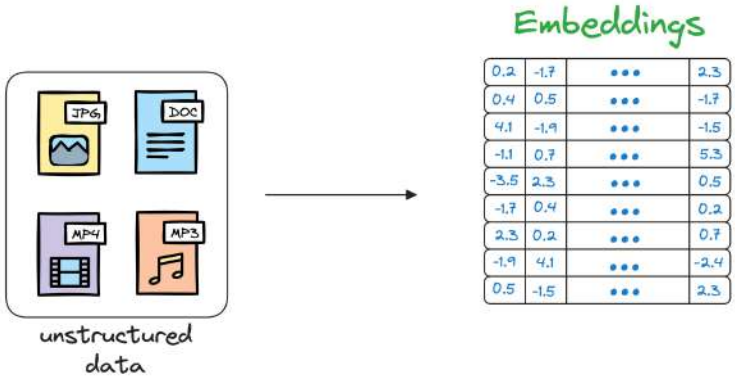
With RAG, the language model can use the retrieved information (which is expected to be reliable) from the vector database to ensure that its responses are grounded in real-world knowledge and context, reducing the likelihood of hallucinations.

This makes the model's responses more accurate, reliable, and contextually relevant, while also ensuring that we don't have to train the LLM repeatedly on new data. This makes the model more "real-time" in its responses.

To understand how RAG actually works in practice, we first need to understand vector databases - the storage layer that powers retrieval.

What are vector databases?

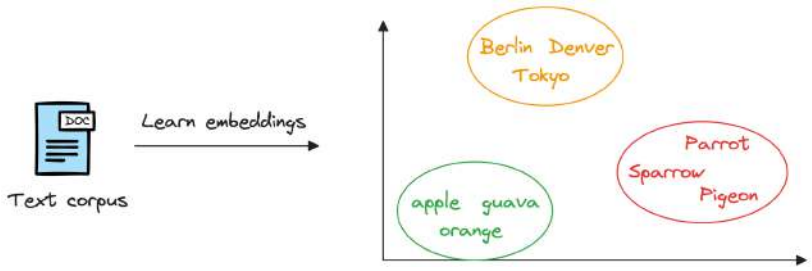
Simply put, a vector database stores unstructured data (text, images, audio, video, etc.) in the form of vector embeddings.



Each data point, whether a word, a document, an image, or any other entity, is transformed into a numerical vector using ML techniques (which we shall see ahead).

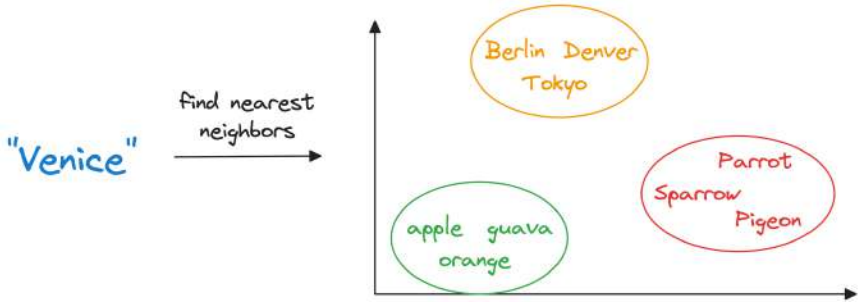
This numerical vector is called an embedding, and the model is trained in such a way that these vectors capture the essential features and characteristics of the underlying data.

Considering word embeddings, for instance, we may discover that in the embedding space, the embeddings of fruits are found close to each other, which cities form another cluster, and so on.



This shows that embeddings can learn the semantic characteristics of entities they represent (provided they are trained appropriately).

Once stored in a vector database, we can retrieve original objects that are similar to the query we wish to run on our unstructured data.



In other words, encoding unstructured data allows us to run many sophisticated operations like similarity search, clustering, and classification over it, which otherwise is difficult with traditional databases.

To exemplify, when an e-commerce website provides recommendations for similar items or searches for a product based on the input query, we're (in most cases) interacting with vector databases behind the scenes.

The purpose of vector databases in RAG

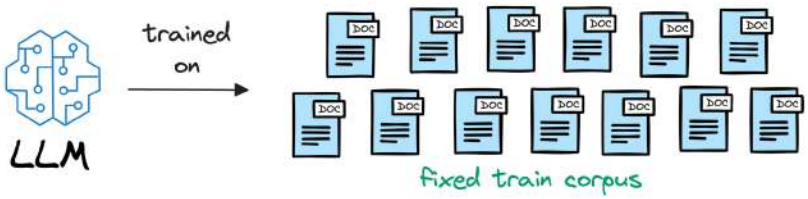
At this point, one interesting thing to learn is how exactly LLMs take advantage of vector databases.

The biggest confusion that people typically face is:

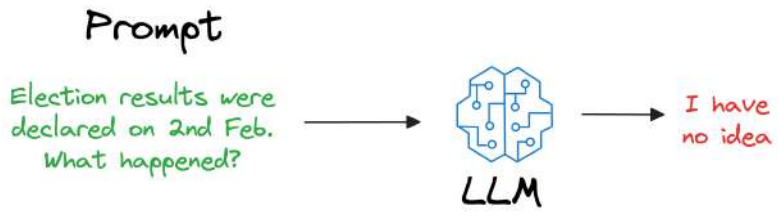
Once we have trained our LLM, it will have some model weights for text generation. Where do vector databases fit in here?

Let's understand this.

To begin, we must understand that an LLM is deployed after learning from a static version of the corpus it was fed during training.



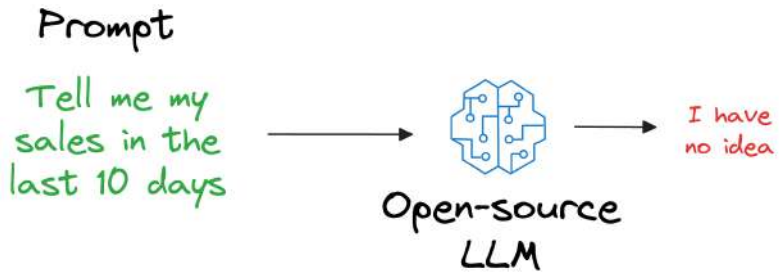
For instance, if the model was deployed after considering the data until 31st Jan 2024, and we use it, say, a week after training, it will have no clue about what happened in those days.



Repeatedly training a new model (or adapting the latest version) every single day on new data is impractical and cost-ineffective. In fact, LLMs can take weeks to train.

Also, what if we open-sourced the LLM and someone else wants to use it on their privately held dataset, which, of course, was not shown during training?

As expected, the LLM will have no clue about it.

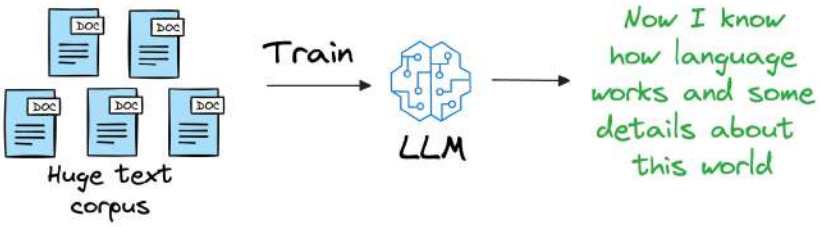


But if you think about it, is it really our objective to train an LLM to know every single thing in the world?

Not at all!

That's not our objective.

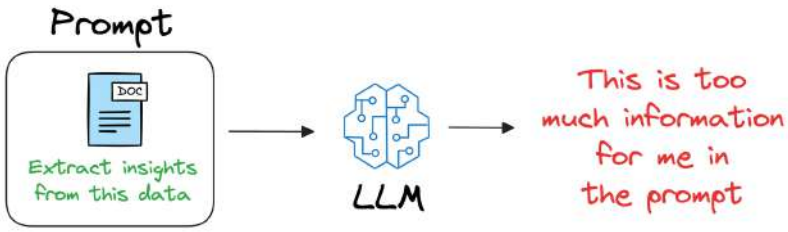
Instead, it is more about helping the LLM learn the overall structure of the language, and how to understand and generate it.



So, once we have trained this model on a ridiculously large enough training corpus, it can be expected that the model will have a decent level of language understanding and generation capabilities.

Thus, if we could figure out a way for LLMs to look up new information they were not trained on and use it in text generation (without training the model again), that would be great!

One way could be to provide that information in the prompt itself.

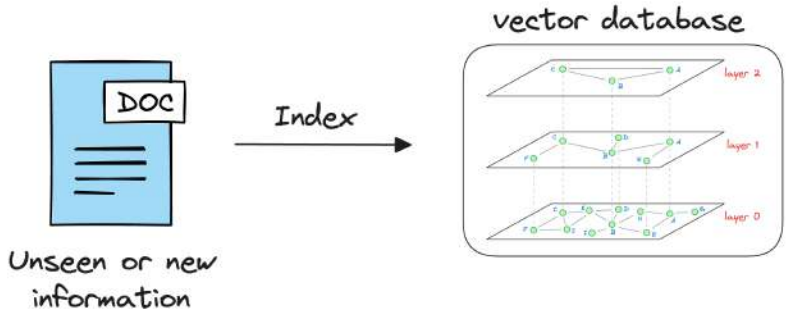


But since LLMs usually have a limit on the context window (number of words/tokens they can accept), the additional information can exceed that limit.

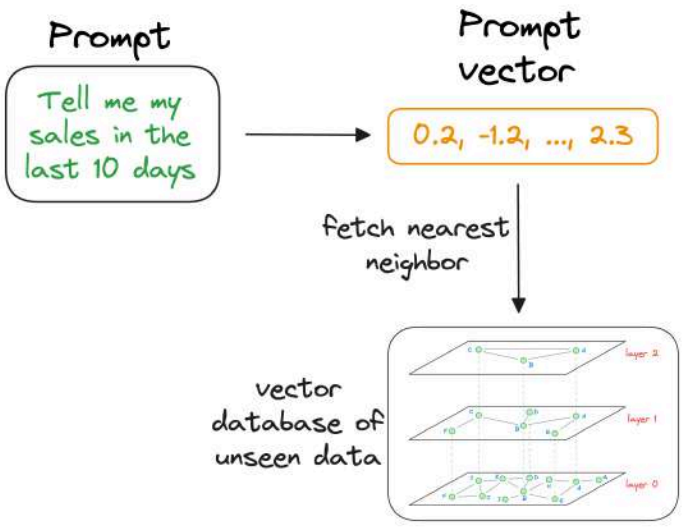
Vector databases solve this problem.

As discussed earlier, vector databases store information in the form of vectors, where each vector captures semantic information about the piece of text being encoded.

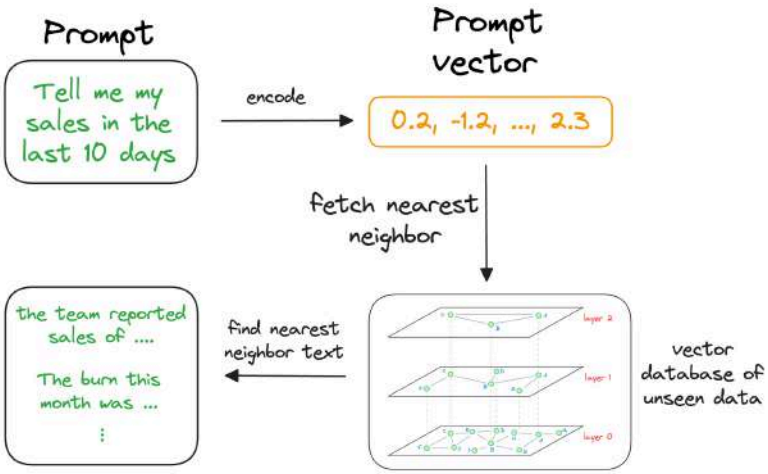
Thus, we can maintain our available information in a vector database by encoding it into vectors using an embedding model.



When the LLM needs to access this information, it can query the vector database using an approximate similarity search with the prompt vector to find content that is similar to the input query vector.

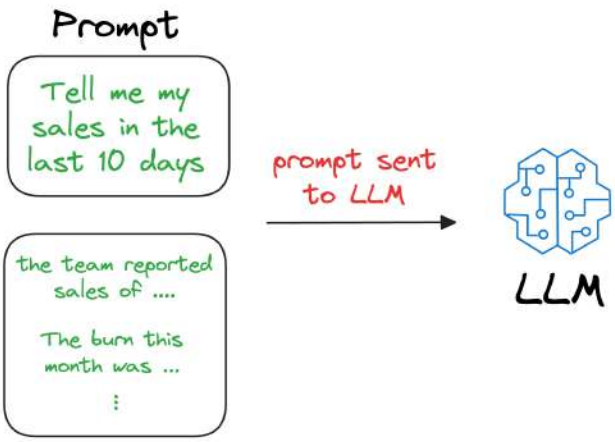


Once the approximate nearest neighbors have been retrieved, we gather the context corresponding to those specific vectors, which were stored at the time of indexing the data in the vector database (this raw data is stored as payload, which we will learn during implementation).



The above search process retrieves context that is similar to the query vector, which represents the context or topic the LLM is interested in.

We can augment this retrieved content along with the actual prompt provided by the user and give it as input to the LLM.



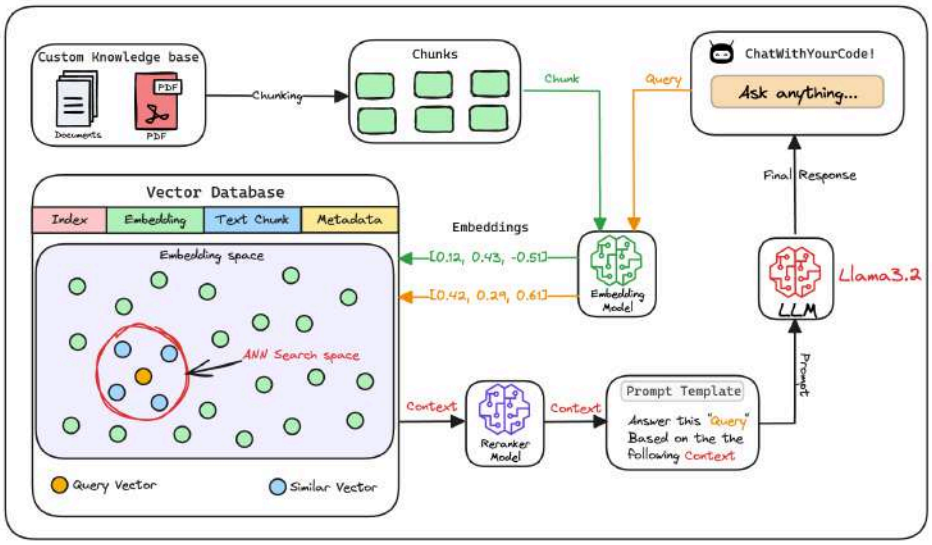
Consequently, the LLM can easily incorporate this info while generating text because it now has the relevant details available in the prompt.

Now that we understand the purpose, let's get into the technical details.

Workflow of a RAG system

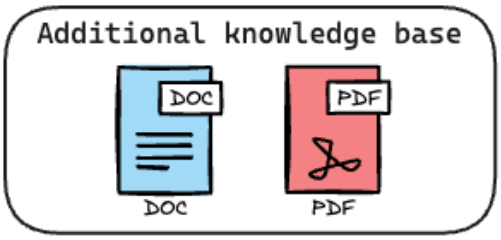
To build a RAG system, it's crucial to understand the foundational components that go into it and how they interact. Thus, in this section, let's explore each element in detail.

Here's an architecture diagram of a typical RAG setup:



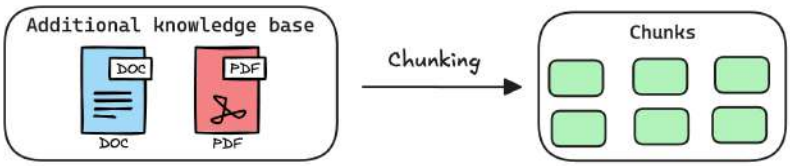
Let's break it down step by step.

We start with some external knowledge that wasn't seen during training, and we want to augment the LLM with:

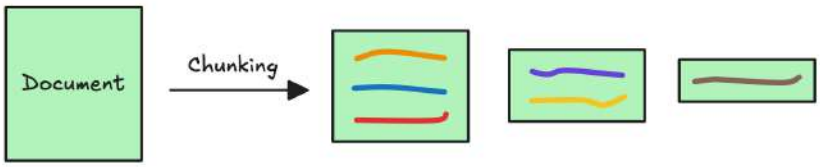


#1) Create chunks

The first step is to break down this additional knowledge into chunks before embedding and storing it in the vector database.



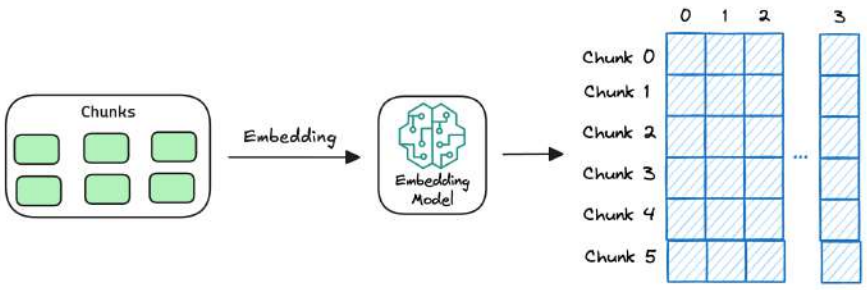
We do this because the additional document(s) can be pretty large. Thus, it is important to ensure that the text fits the input size of the embedding model.



Moreover, if we don't chunk, the entire document will have a single embedding, which won't be of any practical use to retrieve relevant context.

#2) Generate embeddings

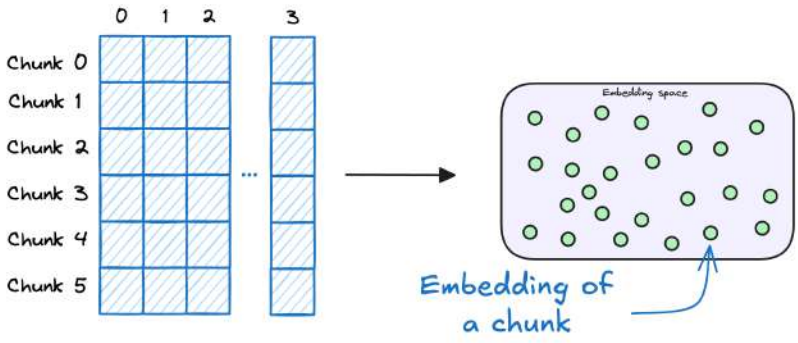
After chunking, we embed the chunks using an embedding model.



Since these are "context embedding models" (not word embedding models), models like bi-encoders are highly relevant here.

#3) Store embeddings in a vector database

These embeddings are then stored in the vector database:



This shows that a vector database acts as a memory for your RAG application since this is precisely where we store all the additional knowledge, using which, the user's query will be answered.

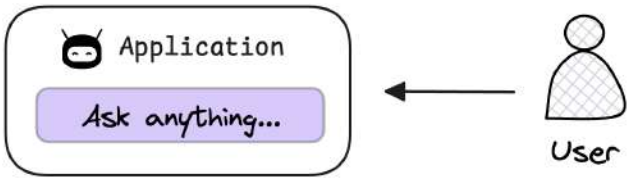
A vector database also stores the metadata and original content along with the vector embeddings.

With that, our vector database has been created and information has been added. More information can be added to this if needed.

Now, we move to the query step.

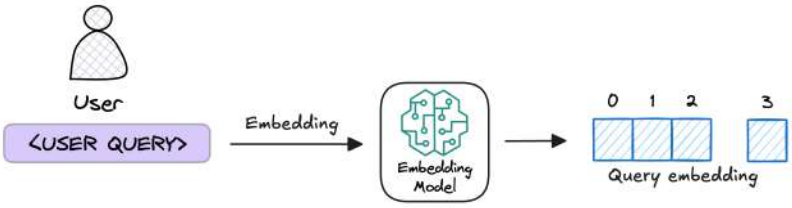
#4) User input query

Next, the user inputs a query, a string representing the information they're seeking.



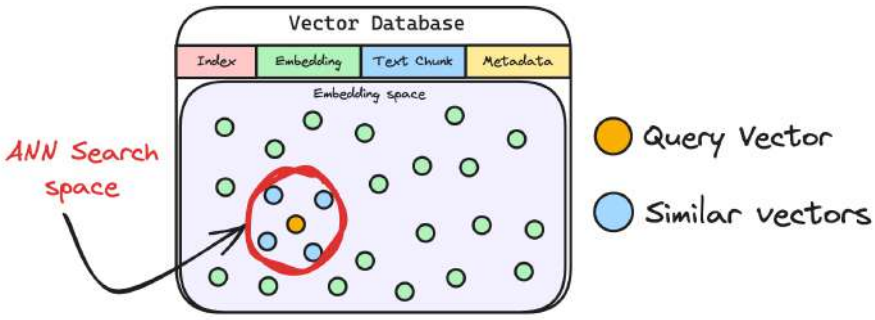
#5) Embed the query

This query is transformed into a vector using the same embedding model we used to embed the chunks earlier in Step 2.

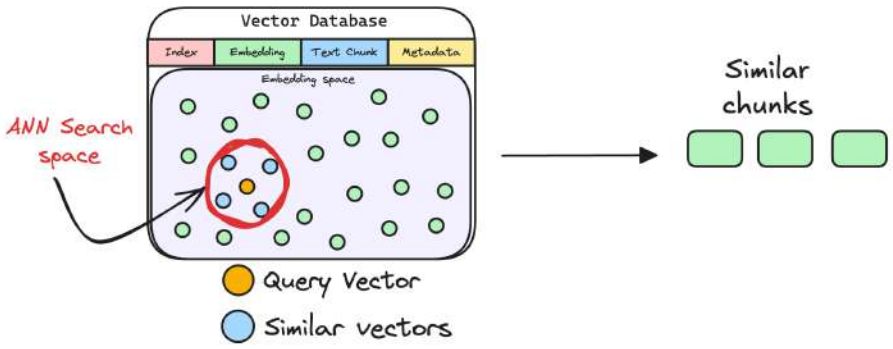


#6) Retrieve similar chunks

The vectorized query is then compared against our existing vectors in the database to find the most similar information.



The vector database returns the k (a pre-defined parameter) most similar documents/chunks (using approximate nearest neighbor search).

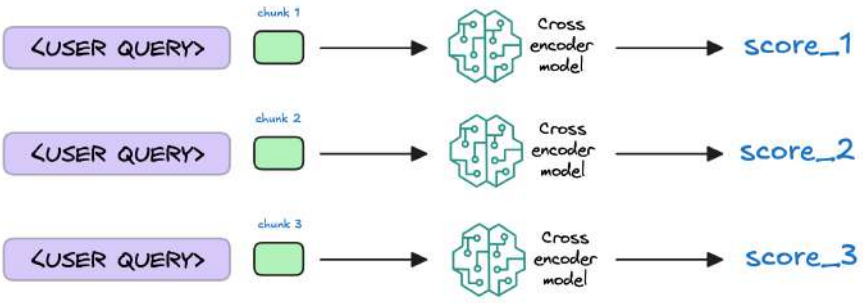


It is expected that these retrieved documents contain information related to the query, providing a basis for the final response generation.

#7) Re-rank the chunks

After retrieval, the selected chunks might need further refinement to ensure the most relevant information is prioritized.

In this re-ranking step, a more sophisticated model (often a cross-encoder) evaluates the initial list of retrieved chunks alongside the query to assign a relevance score to each chunk.



This process rearranges the chunks so that the most relevant ones are prioritized for the response generation.

That said, not every RAG app implements this, and typically, they just rely on the similarity scores obtained in step 6 while retrieving the relevant context from the vector database.

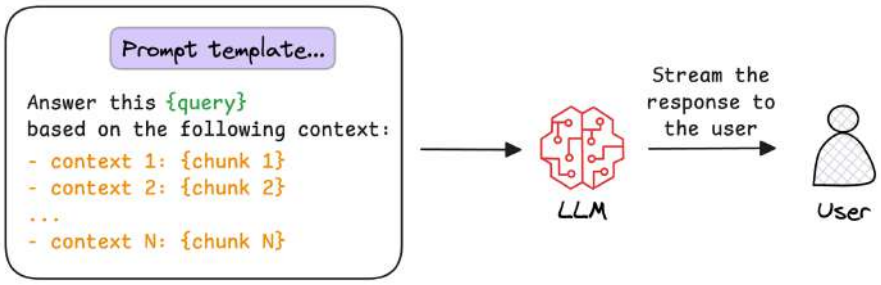
#8) Generate the final response

Almost done!

Once the most relevant chunks are re-ranked, they are fed into the LLM.

This model combines the user's original query with the retrieved chunks in a prompt template to generate a response that synthesizes information from the selected documents.

This is depicted below:

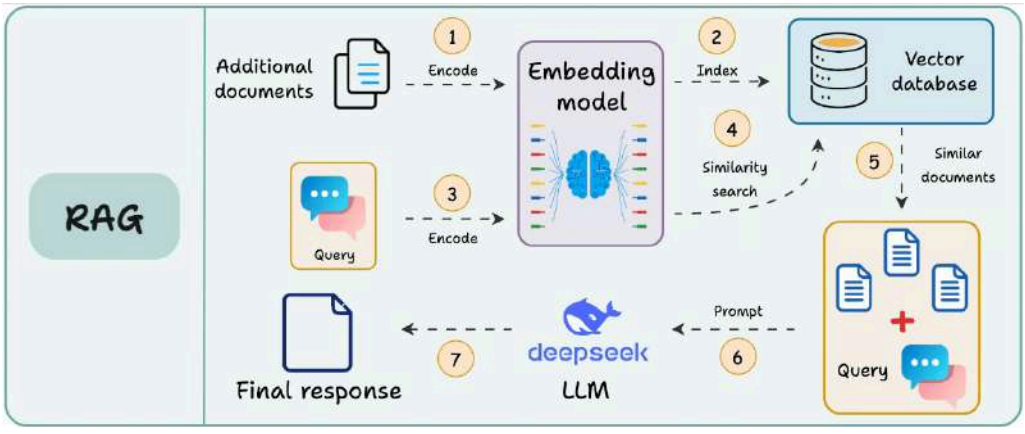


The LLM leverages the context provided by the chunks to generate a coherent and contextually relevant answer that directly addresses the user's query.

Since chunking is the very first step in any RAG pipeline, it's important to understand the different ways it can be done.

5 chunking strategies for RAG

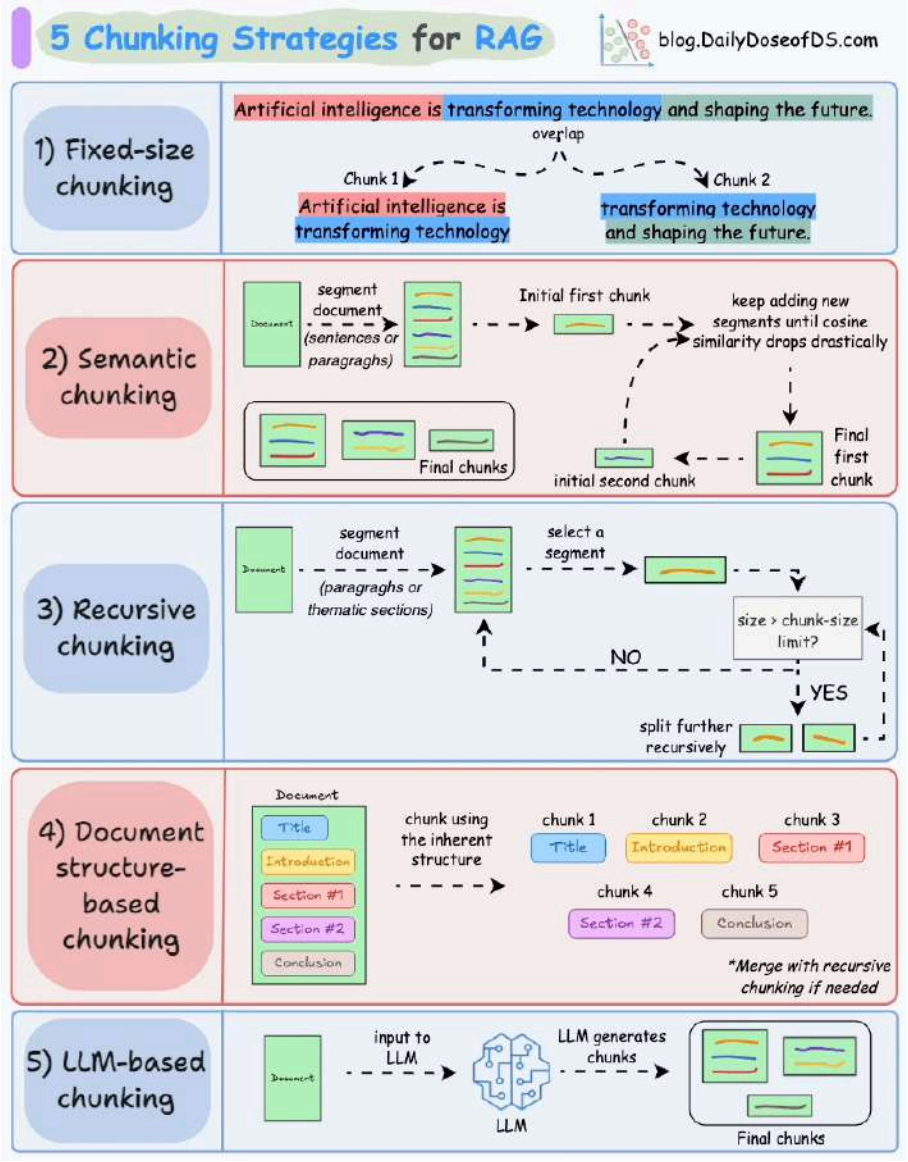
Here's the typical workflow of RAG:



Since the additional document(s) can be large, step 1 also involves chunking, wherein a large document is divided into smaller/manageable pieces.

This step is crucial since it ensures the text fits the input size of the embedding model.

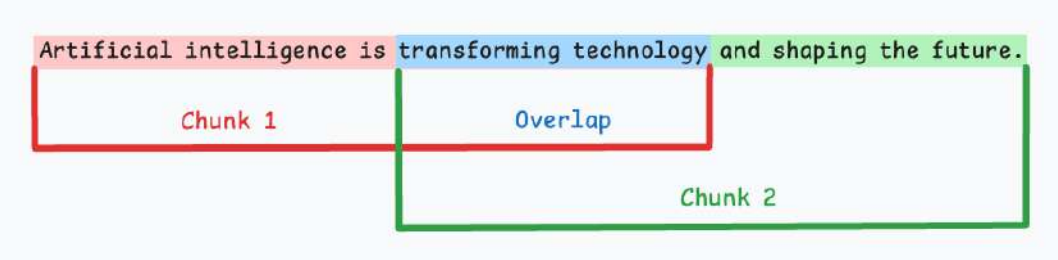
Here are five chunking strategies for RAG:



Let's understand them!

1) Fixed-size chunking

Split the text into uniform segments based on a pre-defined number of characters, words, or tokens.

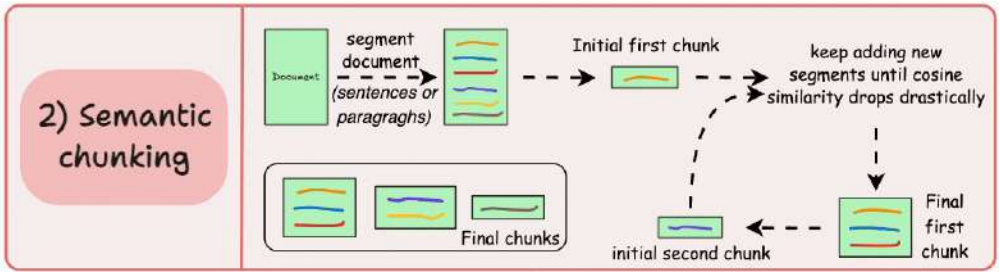


Since a direct split can disrupt the semantic flow, it is recommended to maintain some overlap between two consecutive chunks (the blue part above).

This is simple to implement. Also, since all chunks are of equal size, it simplifies batch processing.

But this usually breaks sentences (or ideas) in between. Thus, important information will likely get distributed between chunks.

2) Semantic chunking



Segment the document based on meaningful units like sentences, paragraphs, or thematic sections.

Next, create embeddings for each segment.

Let's say we start with the first segment and its embedding.

If the first segment's embedding has a high cosine similarity with that of the second segment, both segments form a chunk.

This continues until cosine similarity drops significantly.

The moment it does, we start a new chunk and repeat.

Here's what the output could look like:

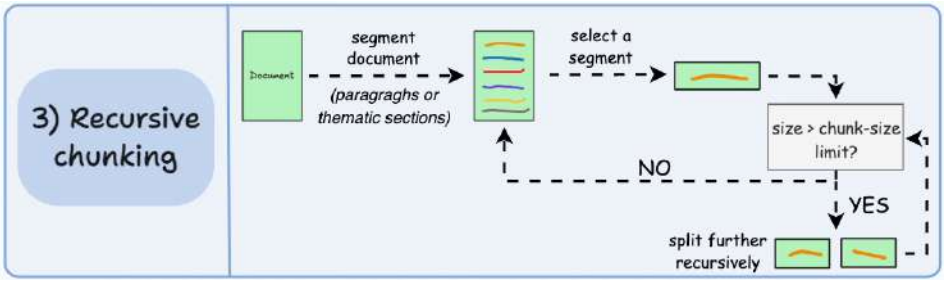
```
Artificial intelligence is transforming industries by automating
processes, enhancing decision-making, and providing insights through
data analysis. Machine learning, a subset of AI, enables systems to learn
and improve from experience without explicit programming. Deep learning,
a branch of machine learning, uses neural networks with multiple layers
to model complex patterns in data.
```

Unlike fixed-size chunks, this maintains the natural flow of language and preserves complete ideas.

Since each chunk is richer, it improves the retrieval accuracy, which, in turn, produces more coherent and relevant responses by the LLM.

A minor problem is that it depends on a threshold to determine if cosine similarity has dropped significantly, which can vary from document to document.

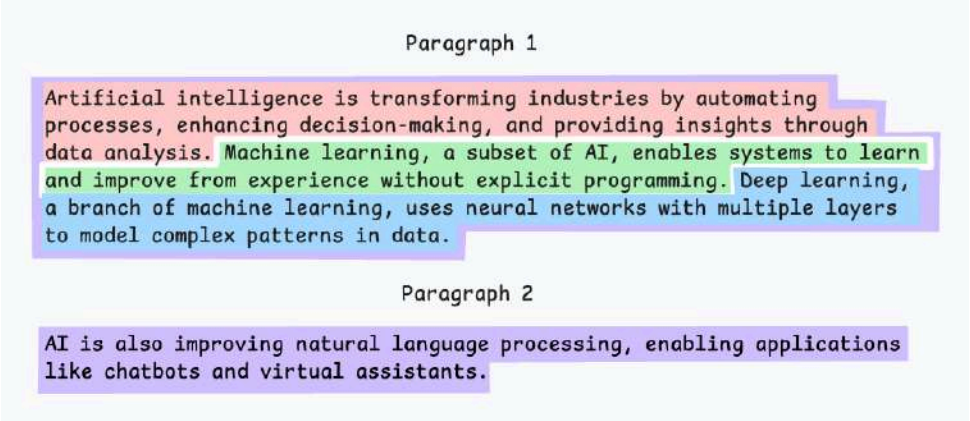
3) Recursive chunking



First, chunk based on inherent separators like paragraphs, or sections.

Next, split each chunk into smaller chunks if the size exceeds a pre-defined chunk size limit. If, however, the chunk fits the chunk-size limit, no further splitting is done.

Here's what the output could look like:



As shown above:

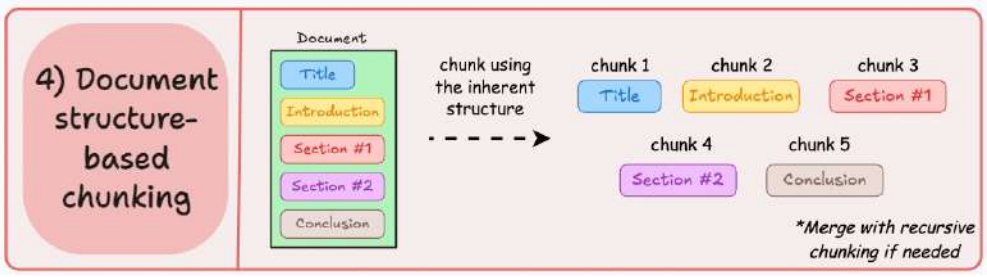
First, we define two chunks (the two paragraphs in purple).

Next, paragraph 1 is further split into smaller chunks.

Unlike fixed-size chunks, this approach also maintains the natural flow of language and preserves complete ideas.

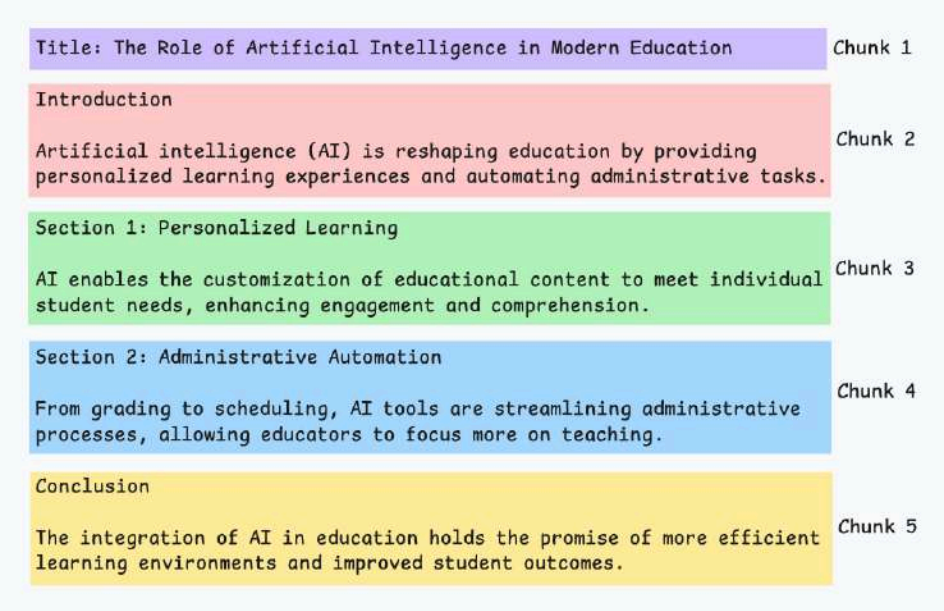
However, there is some extra overhead in terms of implementation and computational complexity.

4) Document structure-based chunking



It utilizes the inherent structure of documents, like headings, sections, or paragraphs, to define chunk boundaries. This way, it maintains structural integrity by aligning with the document's logical sections.

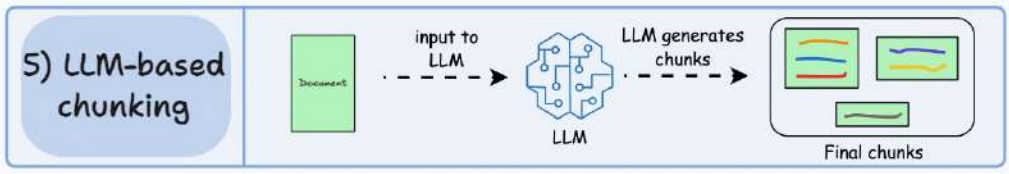
Here's what the output could look like:



That said, this approach assumes that the document has a clear structure, which may not be true.

Also, chunks may vary in length, possibly exceeding model token limits. You can try merging it with recursive splitting.

5) LLM-based chunking



Prompt the LLM to generate semantically isolated and meaningful chunks.

This method ensures high semantic accuracy since the LLM can understand context and meaning beyond simple heuristics (used in the above four approaches).

But this is the most computationally demanding chunking technique of all five techniques discussed here.

Also, since LLMs typically have a limited context window, that is something to be taken care of.

Each technique has its own advantages and trade-offs.

We have observed that semantic chunking works pretty well in many cases, but again, you need to test.

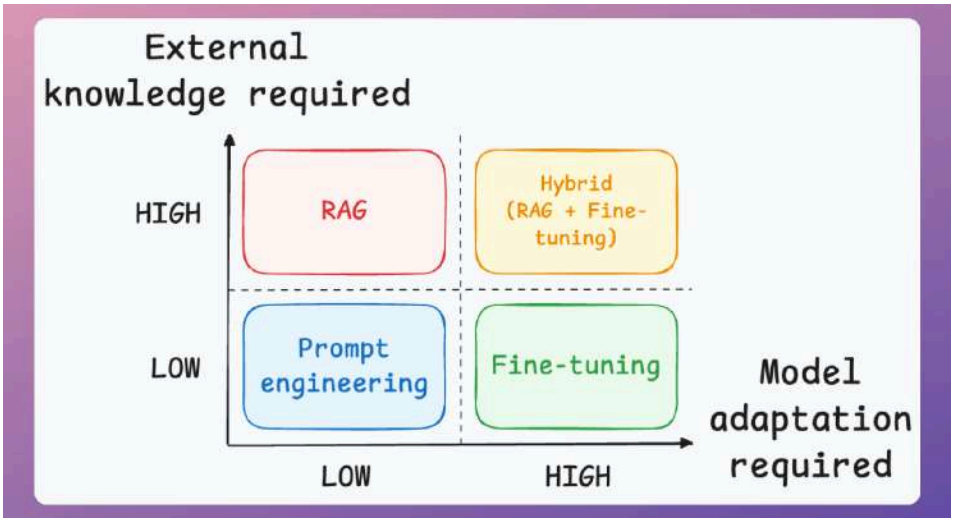
The choice will depend on the nature of your content, the capabilities of the embedding model, computational resources, etc.

Prompting vs. RAG vs. Finetuning?

If you are building real-world LLM-based apps, it is unlikely you can start using the model right away without adjustments. To maintain high utility, you either need:

- Prompt engineering
- Fine-tuning
- RAG
- Or a hybrid approach (RAG + fine-tuning)

The following visual will help you decide which one is best for you:



Two important parameters guide this decision:

- The amount of external knowledge required for your task.
- The amount of adaptation you need. Adaptation, in this case, means changing the behavior of the model, its vocabulary, writing style, etc.

For instance, an LLM might find it challenging to summarize the transcripts of company meetings because speakers might be using some internal vocabulary in their discussions.

So here's the simple takeaway:

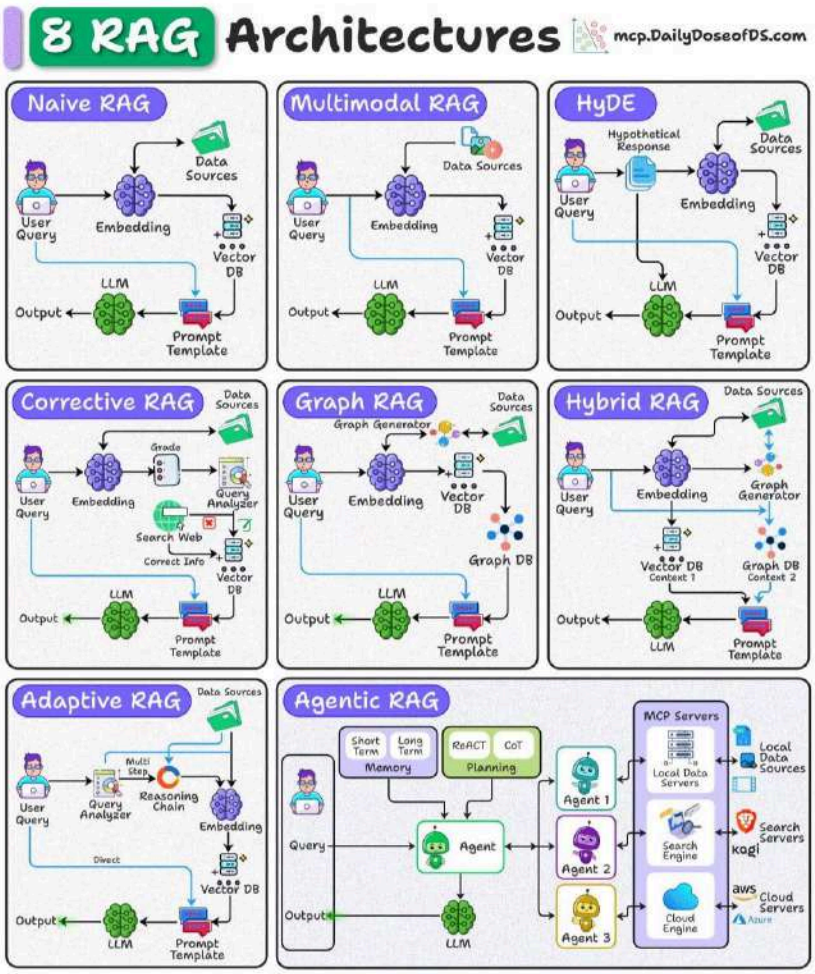
- Use RAGs to generate outputs based on a custom knowledge base if the vocabulary & writing style of the LLM remains the same.
- Use fine-tuning to change the structure (behaviour) of the model than knowledge.
- Prompt engineering is sufficient if you don't have a custom knowledge base and don't want to change the behavior.
- And finally, if your application demands a custom knowledge base and a change in the model's behavior, use a hybrid (RAG + Fine-tuning) approach.

That's it!

Once you've decided that RAG is the right approach, the next step is choosing the right RAG architecture for your use case.

8 RAG architectures

We prepared the following visual that details 8 types of RAG architectures used in AI systems:



Let's discuss them briefly:

1) Naive RAG

Retrieves documents purely based on vector similarity between the query embedding and stored embeddings.

Works best for simple, fact-based queries where direct semantic matching suffices.

2) Multimodal RAG

Handles multiple data types (text, images, audio, etc.) by embedding and retrieving across modalities.

Ideal for cross-modal retrieval tasks like answering a text query with both text and image context.

3) HyDE

Queries are not semantically similar to documents.

This technique generates a hypothetical answer document from the query before retrieval.

Uses this generated document's embedding to find more relevant real documents.

4) Corrective RAG

Validates retrieved results by comparing them against trusted sources (e.g., web search).

Ensures up-to-date and accurate information, filtering or correcting retrieved content before passing to the LLM.

5) Graph RAG

Converts retrieved content into a knowledge graph to capture relationships and entities.

Enhances reasoning by providing structured context alongside raw text to the LLM.

6) Hybrid RAG

Combines dense vector retrieval with graph-based retrieval in a single pipeline.

Useful when the task requires both unstructured text and structured relational data for richer answers.

7) Adaptive RAG

Dynamically decides if a query requires a simple direct retrieval or a multi-step reasoning chain.

Breaks complex queries into smaller sub-queries for better coverage and accuracy.

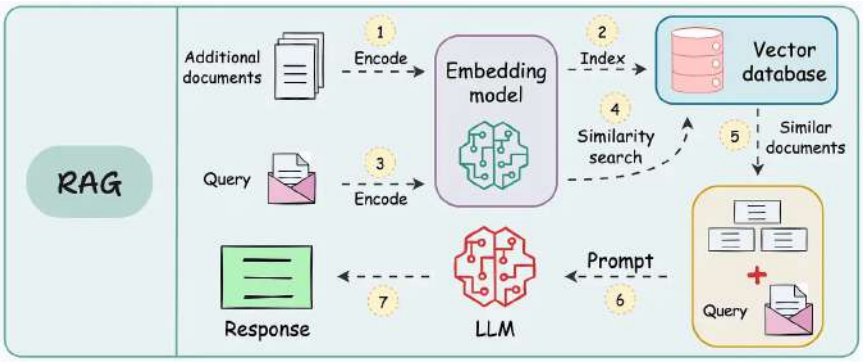
8) Agentic RAG

Uses AI agents with planning, reasoning (ReAct, CoT), and memory to orchestrate retrieval from multiple sources.

Best suited for complex workflows that require tool use, external APIs, or combining multiple RAG techniques.

RAG vs Agentic RAG

These are some issues with the traditional RAG system:



These systems retrieve once and generate once. This means if the retrieved context isn't enough, the LLM can not dynamically search for more information.

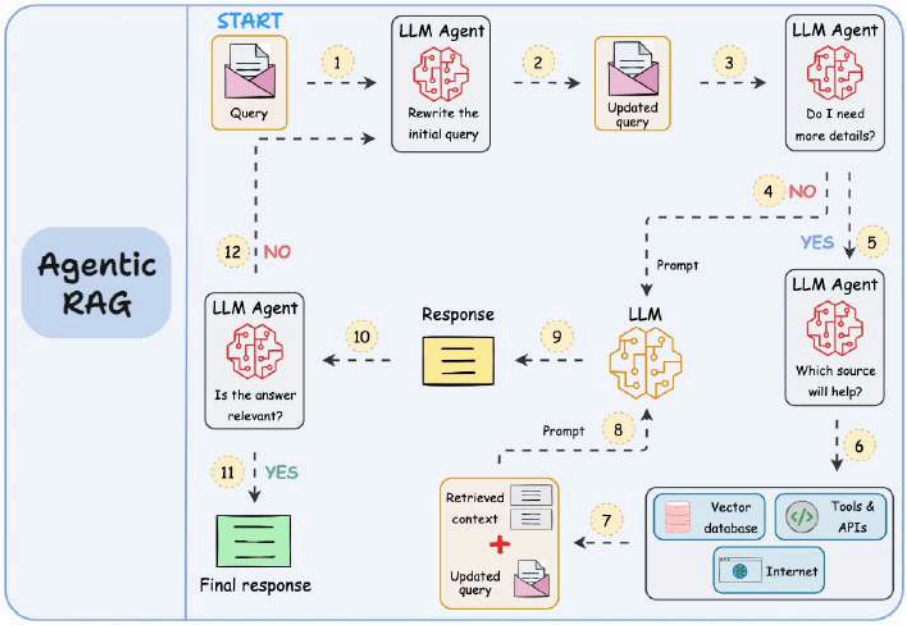
RAG systems may provide relevant context but don't reason through complex queries. If a query requires multiple retrieval steps, traditional RAG falls short.

There's little adaptability. The LLM can't modify its strategy based on the problem at hand.

Due to this, Agentic RAG is becoming increasingly popular. Let's understand this in more detail.

Agentic RAG

The workflow of agentic RAG is depicted below:



Note: The diagram above is one of many blueprints that an agentic RAG system may possess. You can adapt it according to your specific use case.

As shown above, the idea is to introduce agentic behaviors at each stage of RAG.

Think of agents as someone who can actively think through a task - planning, adapting, and iterating until they arrive at the best solution, rather than just

following a defined set of instructions. The powerful capabilities of LLMs make this possible.

Let's understand this step-by-step:

Steps 1-2) The user inputs the query, and an agent rewrites it (removing spelling mistakes, simplifying it for embedding, etc.)

Step 3) Another agent decides whether it needs more details to answer the query.

Step 4) If not, the rewritten query is sent to the LLM as a prompt.

Step 5-8) If yes, another agent looks through the relevant sources it has access to (vector database, tools & APIs, and the internet) and decides which source should be useful. The relevant context is retrieved and sent to the LLM as a prompt.

Step 9) Either of the above two paths produces a response.

Step 10) A final agent checks if the answer is relevant to the query and context.

Step 11) If yes, return the response.

Step 12) If not, go back to Step 1. This procedure continues for a few iterations until the system admits it cannot answer the query.

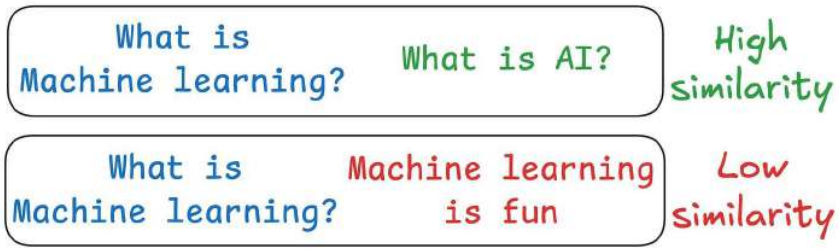
This makes the RAG much more robust since, at every step, agentic behavior ensures that individual outcomes are aligned with the final goal.

That said, it is also important to note that building RAG systems typically boils down to design preferences/choices.

Apart from agentic approaches, another important improvement over traditional RAG comes from better retrieval itself - one popular method being HyDE.

Traditional RAG vs HyDE

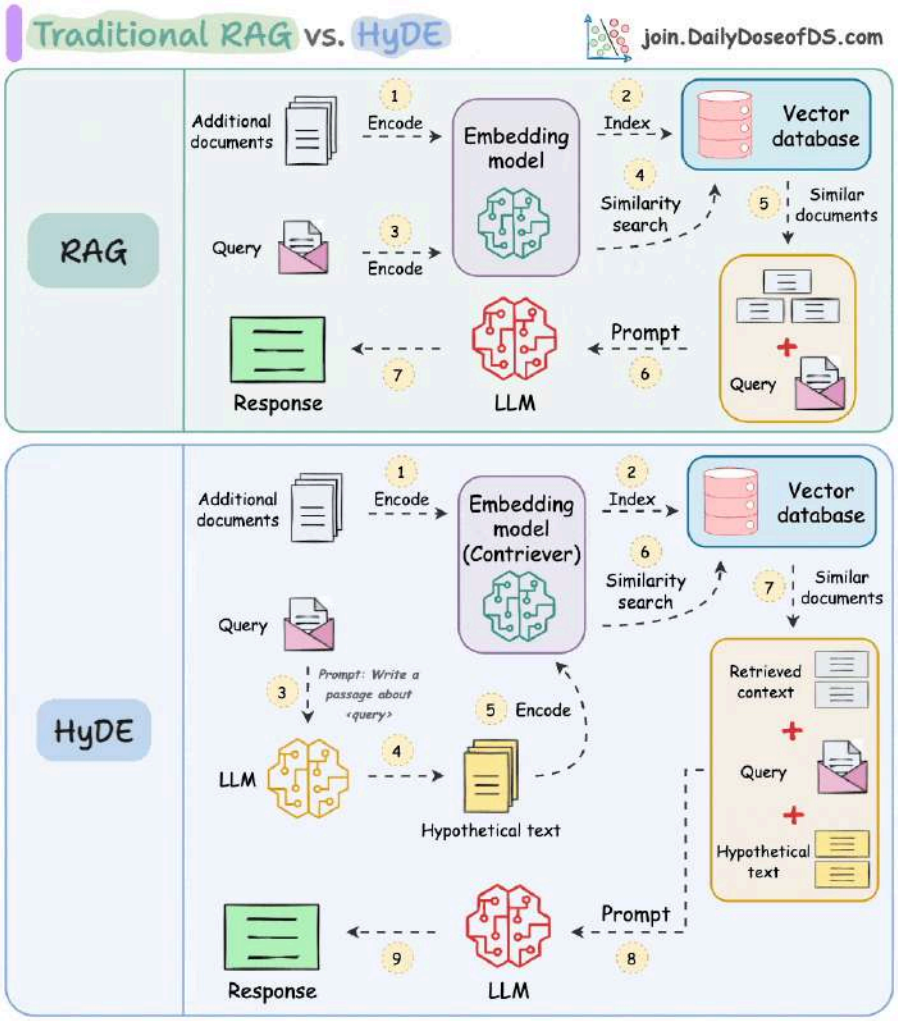
Another critical problem with the traditional RAG system is that questions are not semantically similar to their answers.



As a result, several irrelevant contexts get retrieved during the retrieval step due to a higher cosine similarity than the documents actually containing the answer.

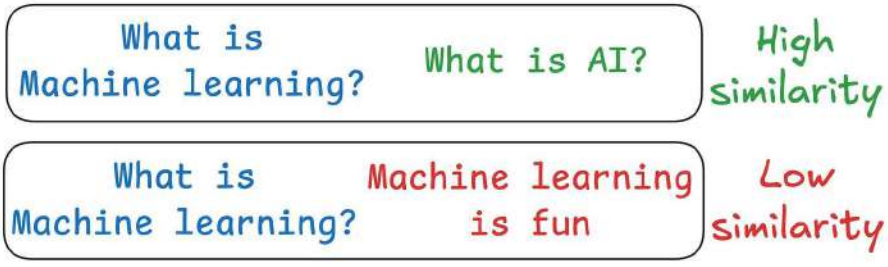
HyDE solves this.

The following visual depicts how it differs from traditional RAG and HyDE.



Let's understand this in more detail.

As mentioned earlier, questions are not semantically similar to their answers, which leads to several irrelevant contexts during retrieval.



HyDE handles this as follows:

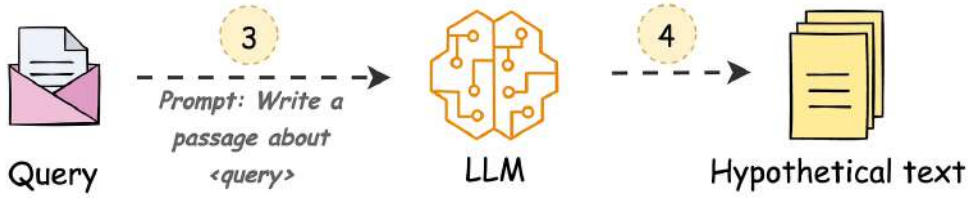
Use an LLM to generate a hypothetical answer H for the query Q (this answer does not have to be entirely correct).

Embed the answer using a contriever model to get E (Bi-encoders are famously used here).

Use the embedding E to query the vector database and fetch relevant context (C).

Pass the hypothetical answer H + retrieved-context C + query Q to the LLM to produce an answer.

Done!



Now, of course, the hypothetical generated will likely contain hallucinated details.

But this does not severely affect the performance due to the contriever model - one which embeds.

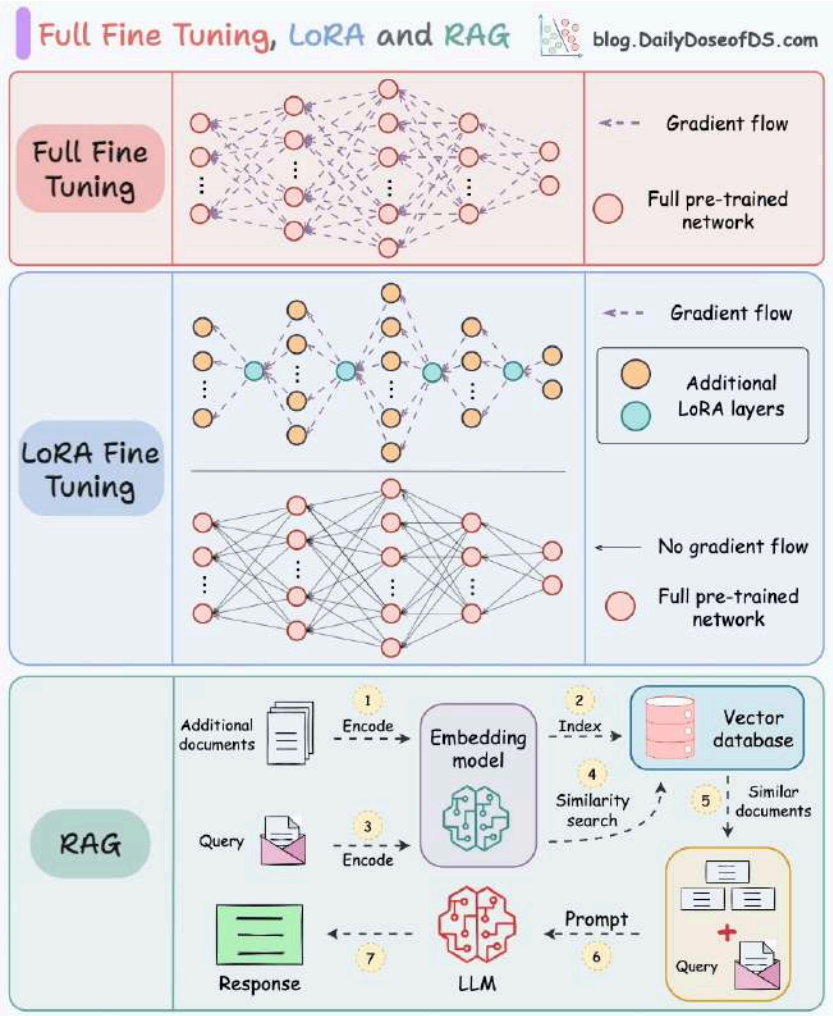
More specifically, this model is trained using contrastive learning and it also functions as a near-lossless compressor whose task is to filter out the hallucinated details of the fake document.

This produces a vector embedding that is expected to be more similar to the embeddings of actual documents than the question is to the real documents:

Several studies have shown that HyDE improves the retrieval performance compared to the traditional embedding model.

But this comes at the cost of increased latency and more LLM usage.

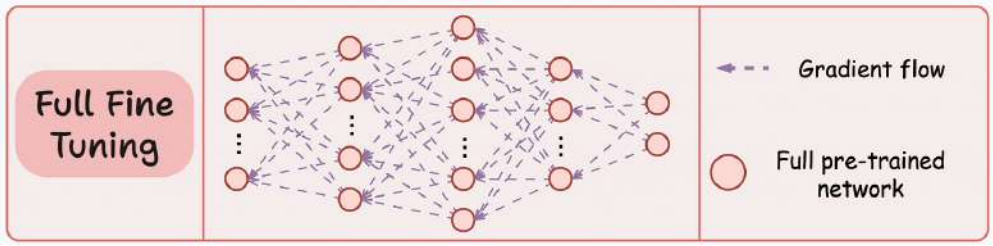
Full-model Fine-tuning vs. LoRA vs. RAG



All three techniques are used to augment the knowledge of an existing model with additional data.

1) Full fine-tuning

Fine-tuning means adjusting the weights of a pre-trained model on a new dataset for better performance.



While this fine-tuning technique has been successfully used for a long time, problems arise when we use it on much larger models — LLMs, for instance, primarily because of:

Their size.

The cost involved in fine-tuning all weights.

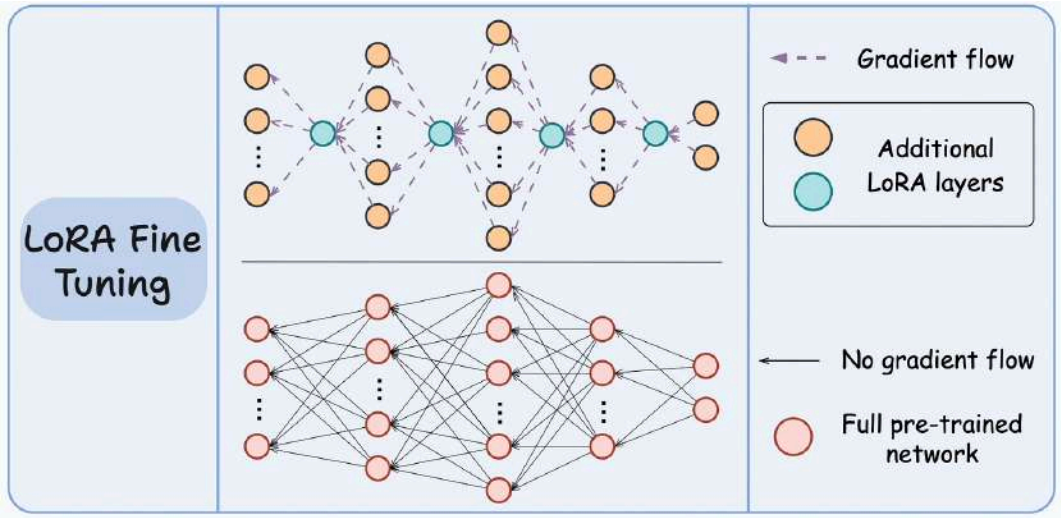
The cost involved in maintaining all large fine-tuned models.

2) LoRA fine-tuning

LoRA fine-tuning addresses the limitations of traditional fine-tuning.

The core idea is to decompose the weight matrices (some or all) of the original model into low-rank matrices and train them instead.

For instance, in the graphic below, the bottom network represents the large pre-trained model, and the top network represents the model with LoRA layers.



The idea is to train only the LoRA network and freeze the large model.

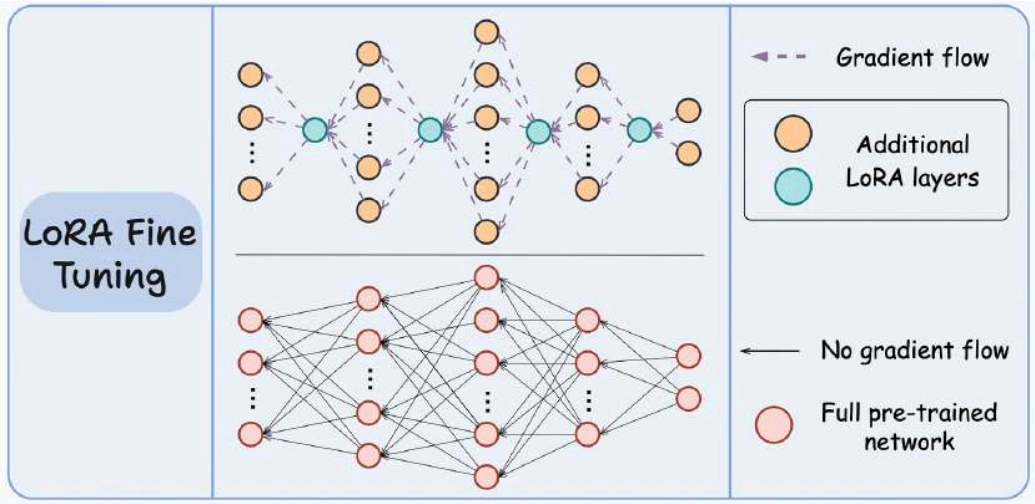
Looking at the above visual, you might think:

But the LoRA model has more neurons than the original model. How does that help?

To understand this, you must make it clear that neurons don't have anything to do with the memory of the network. They are just used to illustrate the dimensionality transformation from one layer to another.

It is the weight matrices (or the connections between two layers) that take up memory.

Thus, we must be comparing these connections instead:

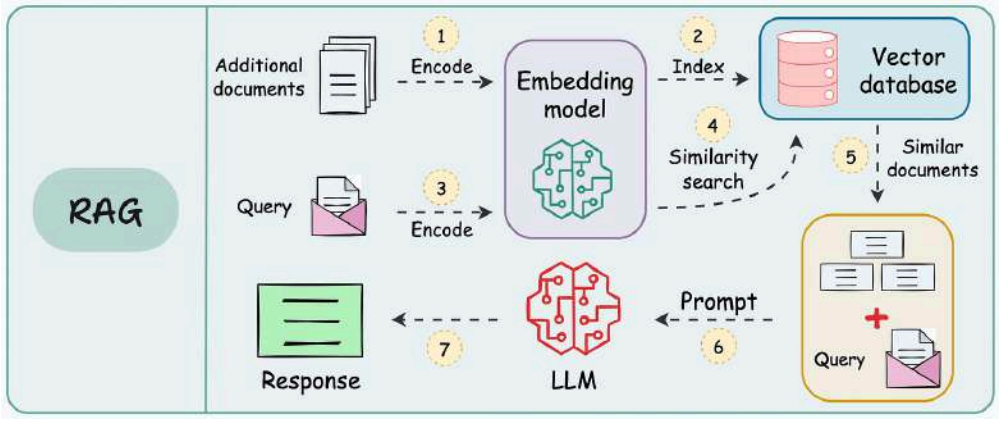


Looking at the above visual, it is pretty clear that the LoRA network has relatively very few connections.

3) RAG

Retrieval augmented generation (RAG) is another pretty cool way to augment neural networks with additional information, without having to fine-tune the model.

This is illustrated below:



There are 7 steps, which are also marked in the above visual:

Step 1-2: Take additional data, and dump it in a vector database after embedding. (This is only done once. If the data is evolving, just keep dumping the

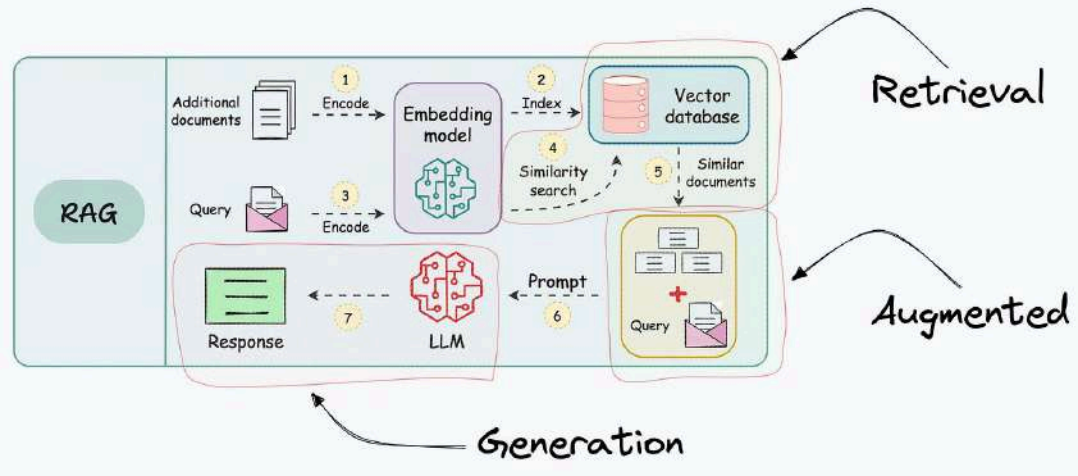
embeddings into the vector database. There's no need to repeat this again for the entire data)

Step 3: Use the same embedding model to embed the user query.

Step 4-5: Find the nearest neighbors in the vector database to the embedded query.

Step 6-7: Provide the original query and the retrieved documents (for more context) to the LLM to get a response.

In fact, even its name entirely justifies what we do with this technique:



Retrieval: Accessing and retrieving information from a knowledge source, such as a database or memory.

Augmented: Enhancing or enriching something, in this case, the text generation process, with additional information or context.

Generation: The process of creating or producing something, in this context, generating text or language.

Of course, there are many problems with RAG too, such as:

RAGs involve similarity matching between the query vector and the vectors of the additional documents. However, questions are structurally very different from answers.

Typical RAG systems are well-suited only for lookup-based question-answering systems. For instance, we cannot build a RAG pipeline to summarize the additional data. The LLM never gets info about all the documents in its prompt because the similarity matching step only retrieves top matches.

So, it's pretty clear that RAG has both pros and cons.

- We never have to fine-tune the model, which saves a lot of computing power.
- But this also limits the applicability to specific types of systems.

RAG vs REFRAG

Most of what we retrieve in RAG setups never actually helps the LLM.

As discussed earlier, in classic RAG, when a query arrives:

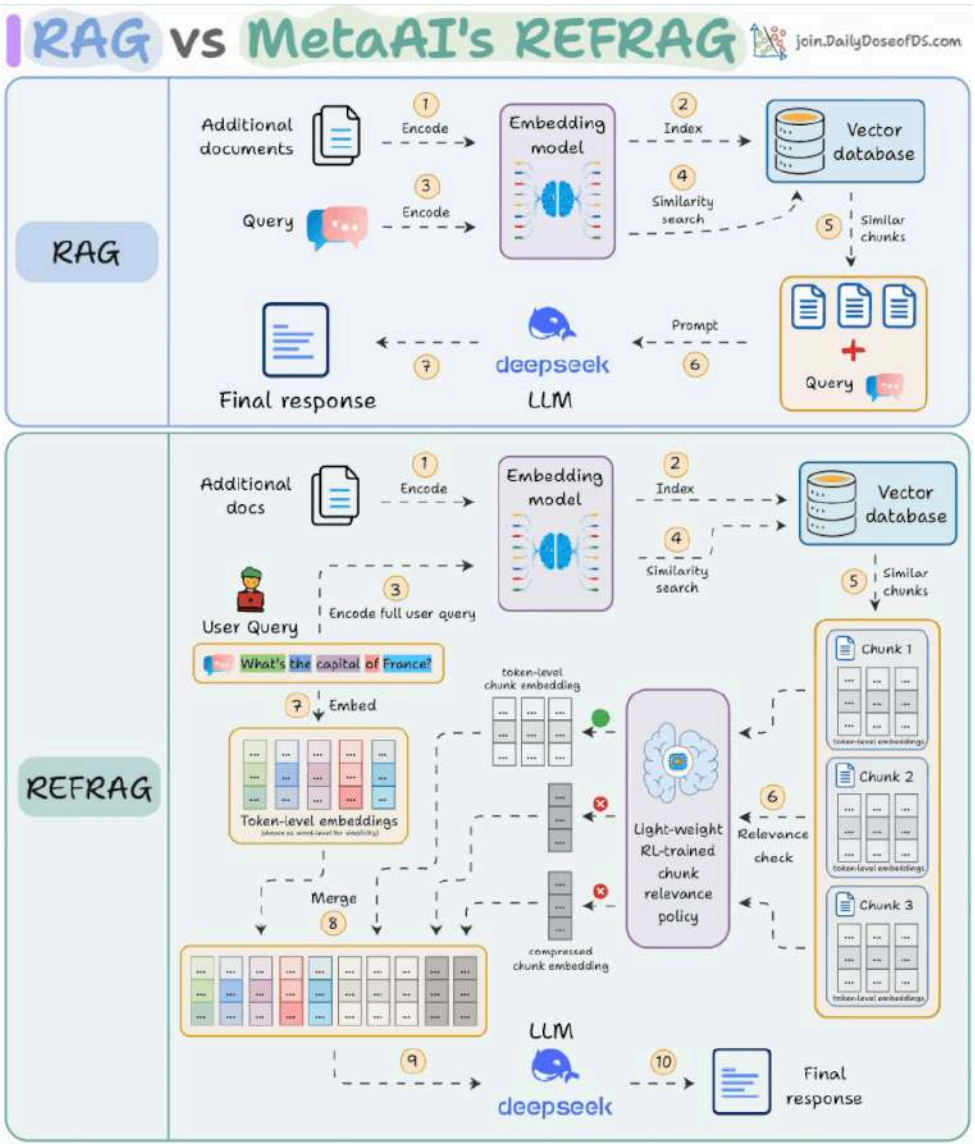
- You encode it into a vector.
- Fetch similar chunks from vector DB.
- Dump the retrieved context into the LLM.

It typically works, but at a huge cost:

- Most chunks contain irrelevant text.
- The LLM has to process far more tokens.
- You pay for compute, latency, and context.

That's the exact problem Meta AI's new method REFRAG solves.

It fundamentally rethinks retrieval and the diagram below explains how it works.



Essentially, instead of feeding the LLM every chunk and every token, REFRAG compresses and filters context at a vector level:

- Chunk compression: Each chunk is encoded into a single compressed embedding, rather than hundreds of token embeddings.
- Relevance policy: A lightweight RL-trained policy evaluates the compressed embeddings and keeps only the most relevant chunks.
- Selective expansion: Only the chunks chosen by the RL policy are expanded back into their full embeddings and passed to the LLM.

This way, the model processes just what matters and ignores the rest.

Here's the step-by-step walkthrough:

Step 1-2) Encode the docs and store them in a vector database.

Step 3-5) Encode the full user query and find relevant chunks. Also, compute the token-level embeddings for both the query (step 7) and matching chunks.

Step 6) Use a relevance policy (trained via RL) to select chunks to keep.

Step 8) Concatenate the token-level representations of the input query with the token-level embedding of selected chunks and a compressed single-vector representation of the rejected chunks.

Step 9-10) Send all that to the LLM.

The RL step makes REFRAG a more relevance-aware RAG pipeline.

Based on the research paper, this approach:

- has 30.85x faster time-to-first-token (3.75x better than previous SOTA)
- provides 16x larger context windows
- outperforms LLaMA on 16 RAG benchmarks while using 2–4x fewer decoder tokens.
- leads to no accuracy loss across RAG, summarization, and multi-turn conversation tasks

That means you can process 16x more context at 30x the speed, with the same accuracy.

RAG vs CAG

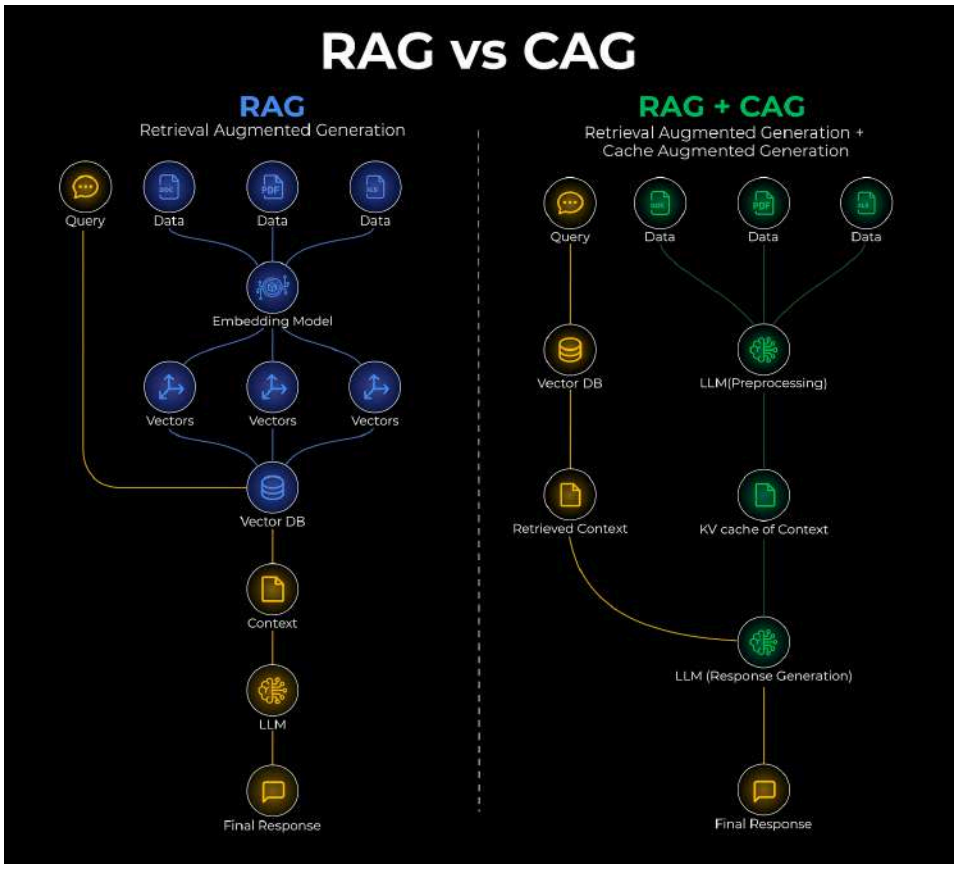
RAG changed how we build knowledge-grounded systems, but it still has a weakness.

Every time a query comes in, the model often re-fetches the same context from the vector DB, which can be expensive, redundant, and slow.

Cache-Augmented Generation (CAG) fixes this.

It lets the model "remember" stable information by caching it directly in the model's key-value memory.

And you can take this one step ahead by fusing RAG and CAG as depicted below:



Here's how it works in simple terms:

- In a regular RAG setup, your query goes to the vector database, retrieves relevant chunks, and feeds them to the LLM.
- But in RAG + CAG, you divide your knowledge into two layers.
    - The static, rarely changing data, like company policies or reference guides, gets cached once inside the model's KV memory.

- The dynamic, frequently updated data, like recent customer interactions or live documents, continues to be fetched via retrieval.

This way, the model doesn't have to reprocess the same static information every time.

It uses it instantly from cache, and supplements it with whatever's new via retrieval to give faster inference.

The key here is to be selective about what you cache.

You should only include stable, high-value knowledge that doesn't change often.

If you cache everything, you'll hit context limits, so separating "cold" (cacheable) and "hot" (retrievable) data is what keeps this system reliable.

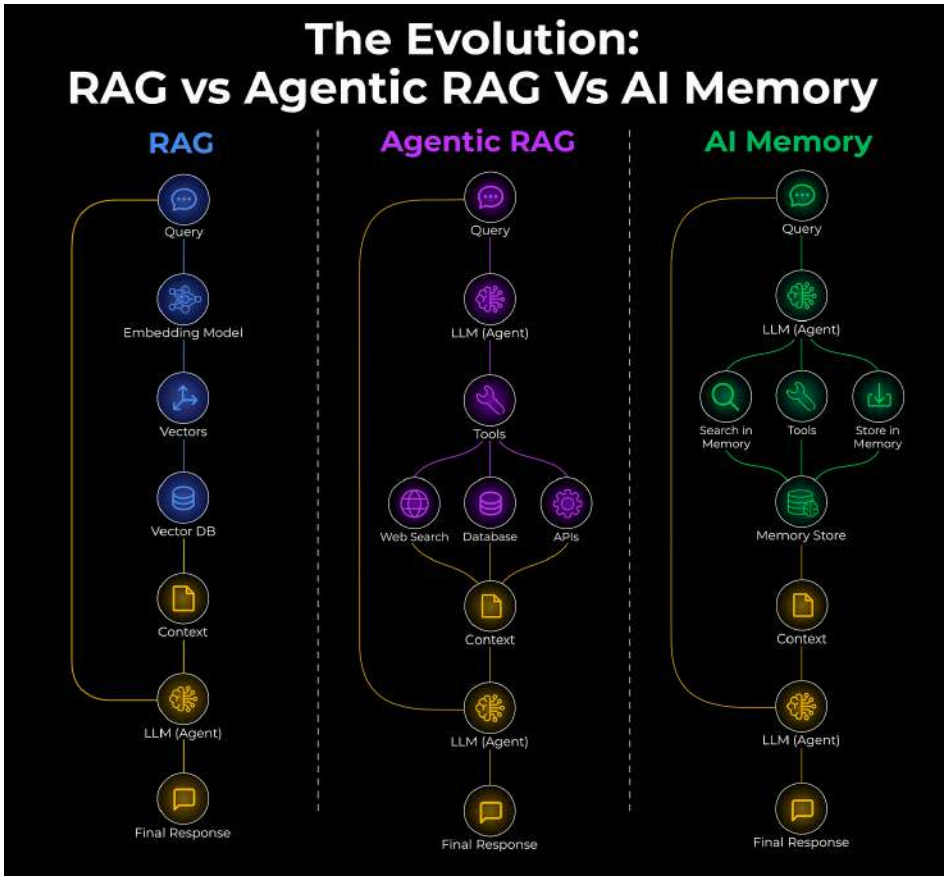
You can also see this in practice above.

Many APIs like OpenAI and Anthropic already support prompt caching, so you can start experimenting right away.

RAG, Agentic RAG and AI Memory

RAG was never the end goal. Memory in AI agents is where everything is heading.

Let's break down this evolution in the simplest way possible.



RAG (2020-2023):

- Retrieve info once, generate response
- No decision-making, just fetch and answer
- Problem: Often retrieves irrelevant context

Agentic RAG:

- Agent decides *if* retrieval is needed
- Agent picks *which* source to query
- Agent validates *if* results are useful
- Problem: Still read-only, can't learn from interactions

AI Memory:

- Reads AND writes to external knowledge
- Learns from past conversations

- Remembers user preferences, past context
- Enables true personalization

The mental model is simple:

- RAG: read-only, one-shot
- Agentic RAG: read-only via tool calls
- Agent Memory: read-write via tool calls

Here's what makes agent memory powerful:

The agent can now "remember" things, like user preferences, past conversations and important dates. All stored and retrievable for future interactions.

This unlocks something bigger: continual learning.

Instead of being frozen at training time, agents can now accumulate knowledge from every interaction. They improve over time without retraining.

Memory is the bridge between static models and truly adaptive AI systems.

Context Engineering

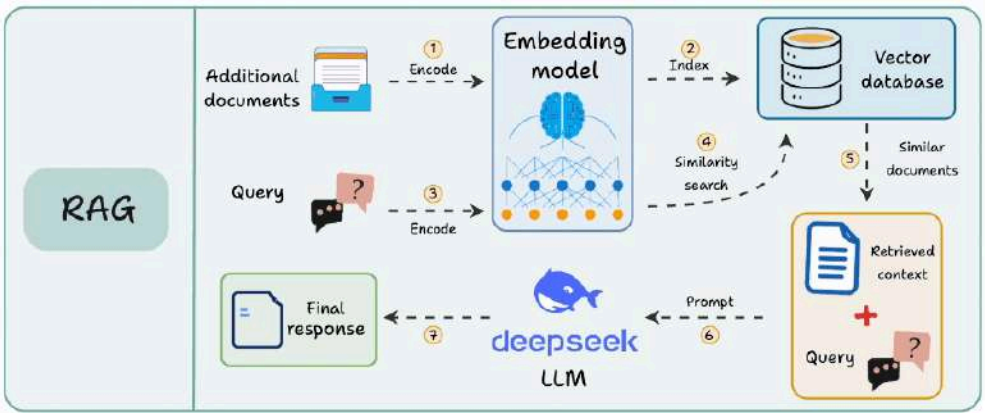
What is Context Engineering?

Context engineering is rapidly becoming a crucial skill for AI engineers. It's no longer just about clever prompting, it's about the systematic orchestration of context.

Here's the current problem:

Most AI agents (or LLM apps) fail not because the models are bad, but because they lack the right context to succeed.

For instance, a RAG workflow is typically 80% retrieval and 20% generation.

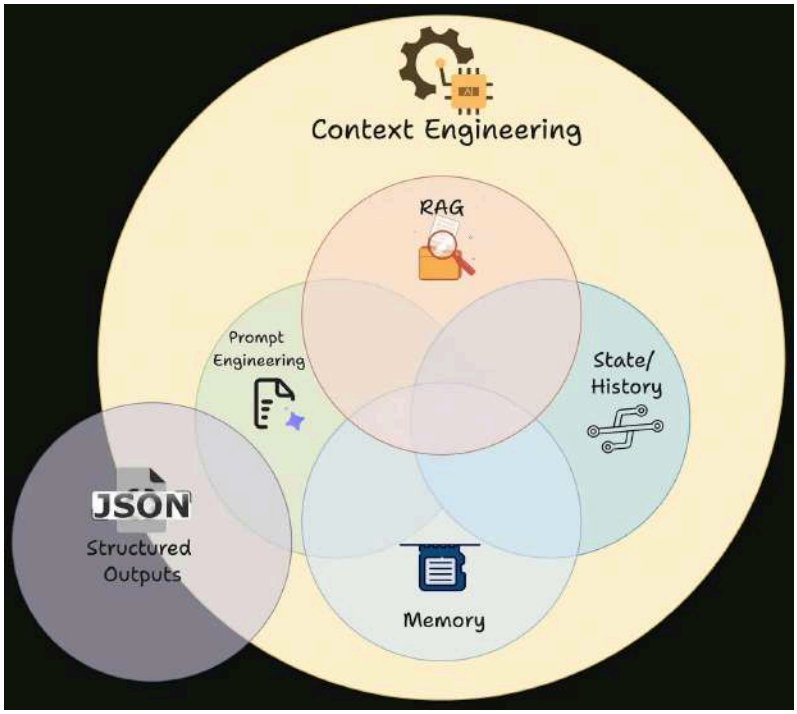


Thus:

- Good retrieval could still work with a weak LLM.
- But bad retrieval can NEVER work even with the best of LLMs.

If your RAG isn't working, most likely, it's a context retrieval issue.

In the same way, LLMs aren't mind readers. They can only work with what you give them.



Context engineering involves creating dynamic systems that offer:

- The right information
- The right tools
- In the right format

This ensures the LLM can effectively complete the task.

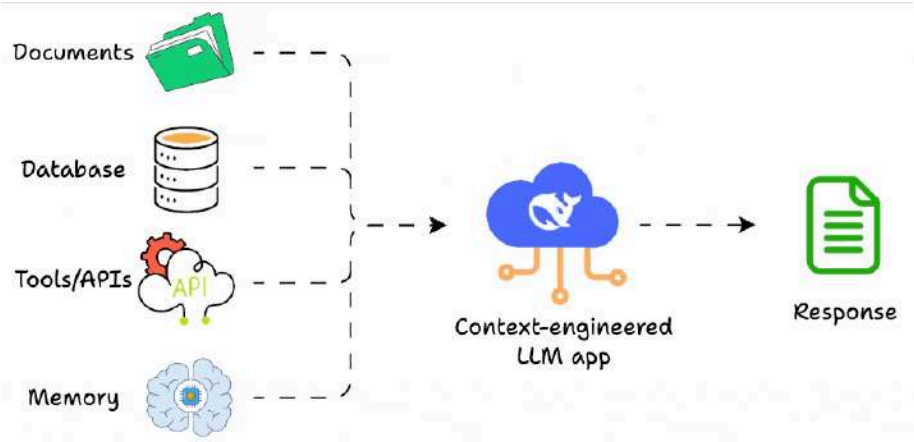
But why was traditional prompt engineering not enough?

Prompt engineering primarily focuses on "magic words" with an expectation of getting a better response.

But as AI applications grow complex, complete and structured context matters far more than clever phrasing.

These are the 4 key components of a context engineering system:

Dynamic information flow: Context comes from multiple sources: users, previous interactions, external data, and tool calls. Your system needs to pull it all together intelligently.



Smart tool access: If your AI needs external information or actions, give it the right tools. Format the outputs so they're maximally digestible.

Memory management:

- Short-term: Summarize long conversations
- Long-term: Remember user preferences across sessions

Format optimization: A short, descriptive error message beats a massive JSON blob every time.

The bottom line is…

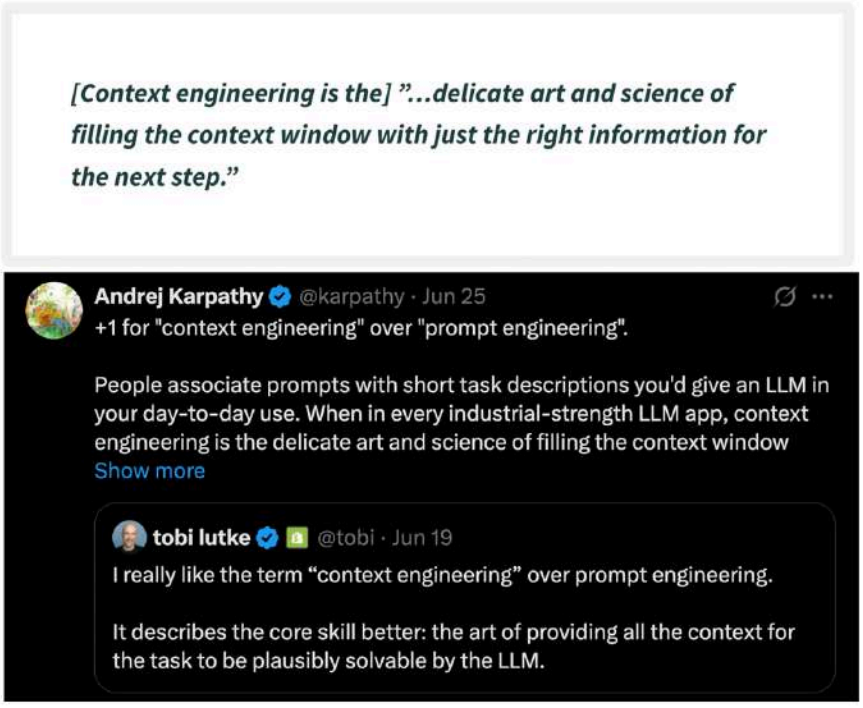
Context engineering is becoming the new core skill since it addresses the real bottleneck, which is not model capability, but setting up an architecture of information.

As models get better, context quality becomes the limiting factor.

Context Engineering for Agents

Simply put, context engineering is the art and science of delivering the right information, in the right format, at the right time, to your LLM.

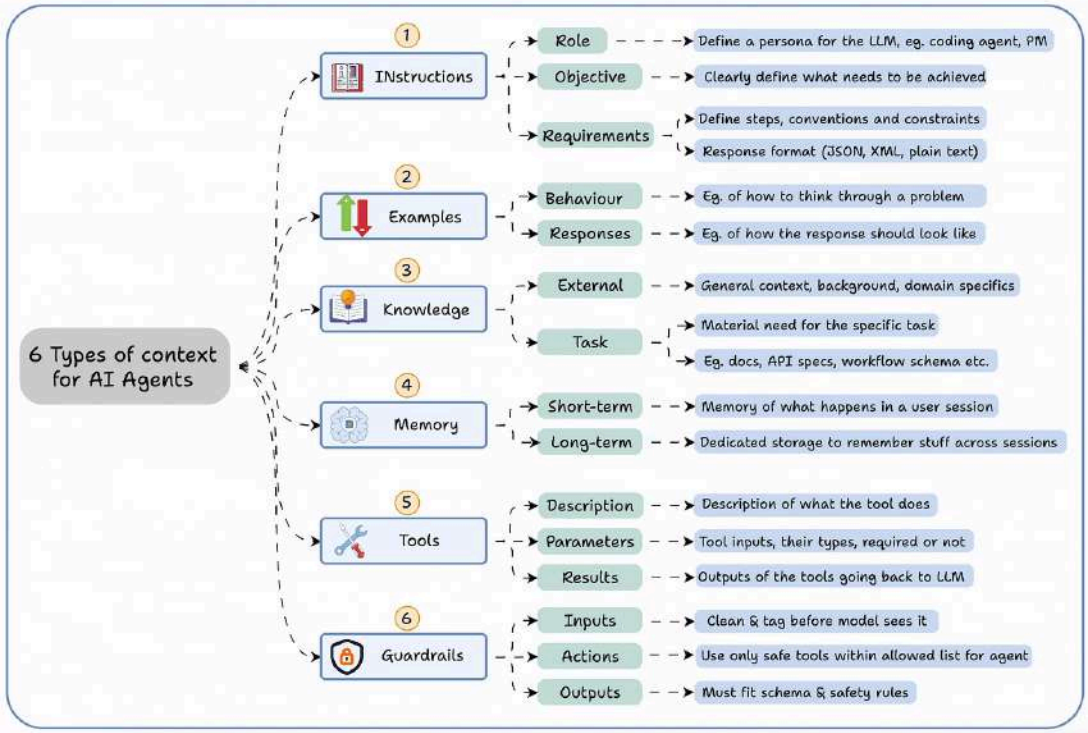
Here's a quote by Andrej Karpathy on context engineering...



To understand context engineering, it's essential to first understand the meaning of context.

Agents today have evolved into much more than just chatbots.

The graphic below summarizes the 6 types of contexts an agent needs to function properly, which are:

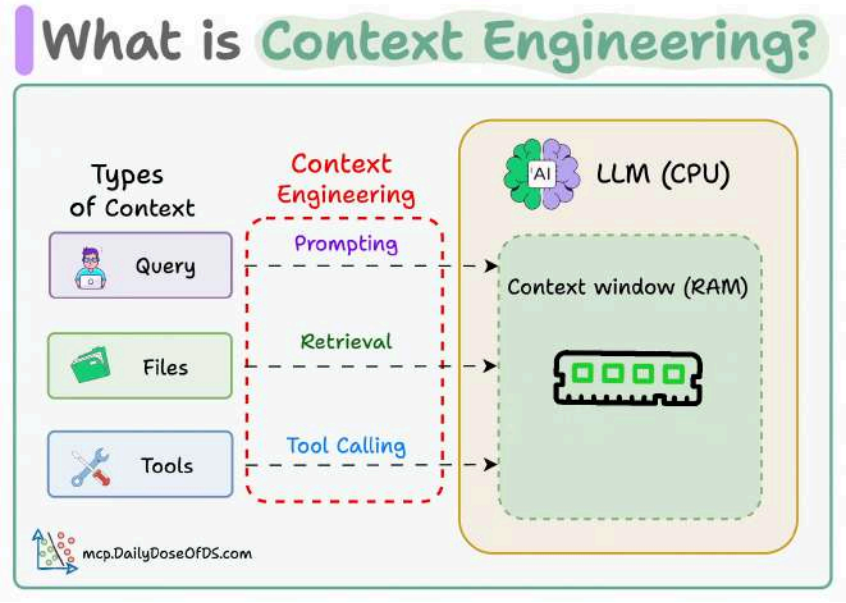


- Instructions
- Examples
- Knowledge
- Memory
- Tools
- Guardrails

This tells you that it's not enough to simply "prompt" the agents.

You must engineer the input (context).

Think of it this way:

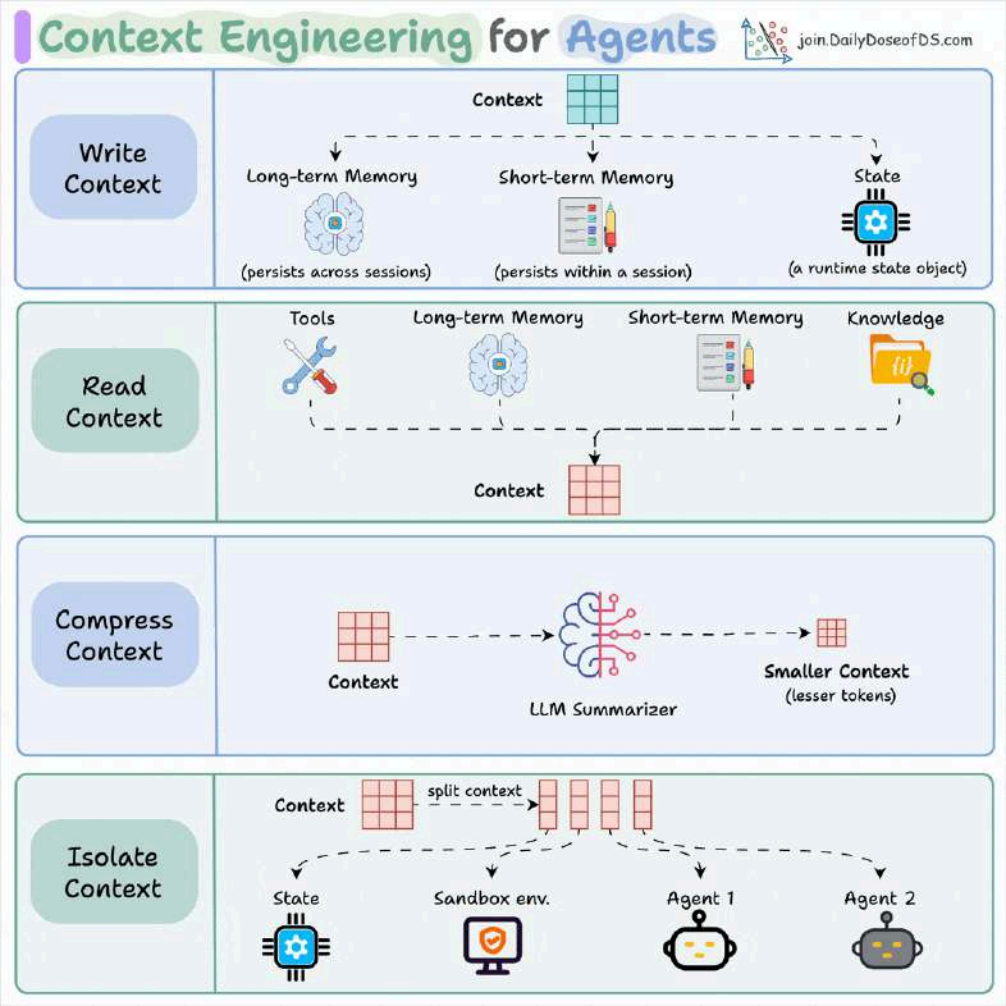


- If LLM is a CPU.
- Then the context window is the RAM.

You're essentially programming the "RAM" with the perfect instructions for your AI.

How do we do it?

Context engineering can be broken down into 4 fundamental stages:

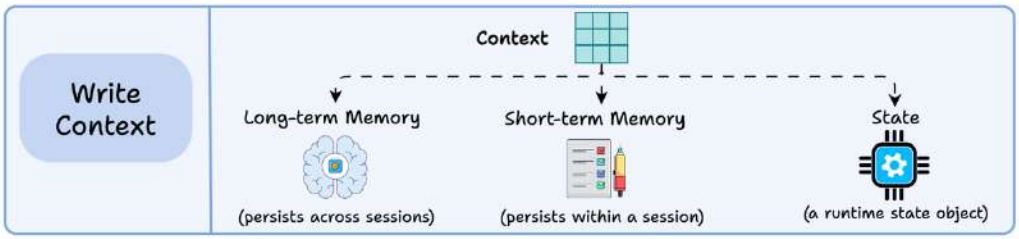


- Writing Context
- Selecting Context
- Compressing Context
- Isolating Context

Let's understand each, one-by-one...

1) Writing context

Writing context means saving it outside the context window to help an agent perform a task.

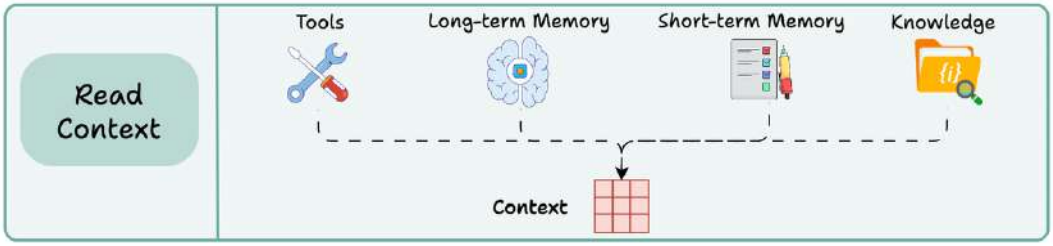


You can do so by writing it to:

- Long-term memory (persists across sessions)
- Short-term memory (persists within a session)
- A state object

2) Read context

Reading context means pulling it into the context window to help an agent perform a task.

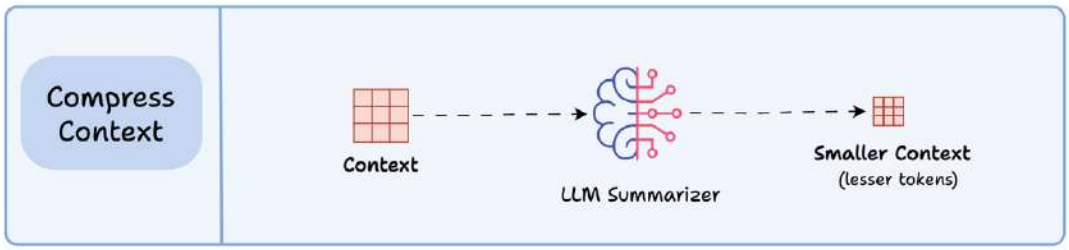


Now this context can be pulled from:

- A tool
- Memory
- Knowledge base (docs, vector DB)

3) Compressing context

Compressing context means keeping only the tokens needed for a task.

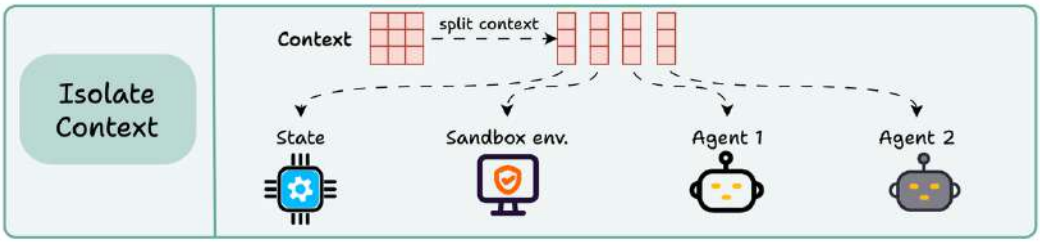


The retrieved context may contain duplicate or redundant information (multi-turn tool calls), leading to extra tokens & increased cost.

Context summarization helps here.

4) Isolating context

Isolating context involves splitting it up to help an agent perform a task.



Some popular ways to do so are:

- Using multiple agents (or sub-agents), each with its own context
- Using a sandbox environment for code storage and execution
- And using a state object

So essentially, when you are building a context engineering workflow, you are engineering a "context" pipeline so that the LLM gets to see the right information, in the right format, at the right time.

This is exactly how context engineering works!

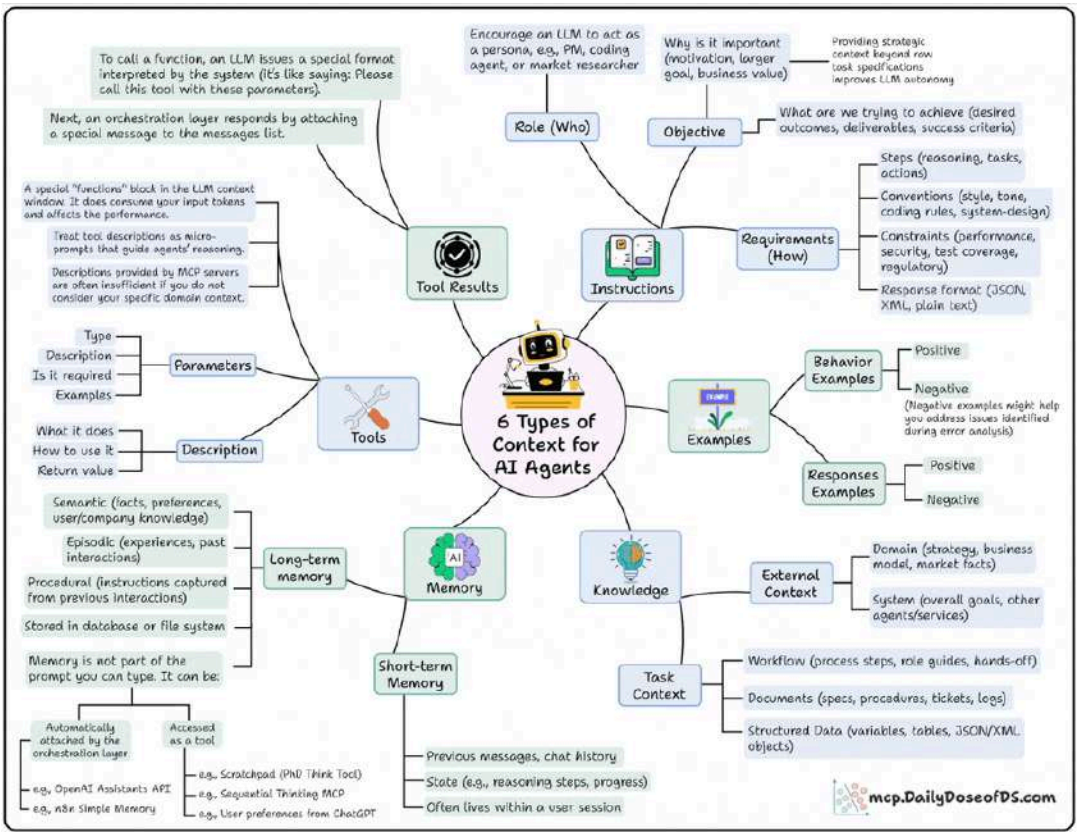
6 Types of Contexts for AI Agents

A poor LLM can possibly work with an appropriate context, but a SOTA LLM can never make up for an incomplete context.

That is why production-grade LLM apps don't just need instructions but rather structure, which is the full ecosystem of context that defines their reasoning, memory, and decision loops.

And all advanced agent architectures now treat context as a multi-dimensional design layer, not a line in a prompt.

Here's the mental model to use when you think about the types of contexts for Agents:



1) Instructions

This defines the who, why, and how:

- Who's the agent? (PM, researcher, coding assistant)
- Why is it acting? (goal, motivation, outcome)
- How should it behave? (steps, tone, format, constraints)

2) Examples

This shows what good and bad look like:

- This includes behavioral demos, structured examples, or even anti-patterns.
- Models learn patterns much better than plain rules

3) Knowledge

This is where you feed it domain knowledge.

- From business processes and APIs to data models and workflows
- This bridges the gap between text prediction and decision-making

4) Memory

You want your Agent to remember what it did in the past. This layer gives it continuity across sessions.

- Short-term: current reasoning steps, chat history
- Long-term: facts, company knowledge, user preferences

5) Tools

This layer extends the Agent's power beyond language and takes real-world action.

- Each tool has parameters, inputs, and examples.
- The design here decides how well your agent uses external APIs.

6) Tool Results

- This layer feeds the tool's results back to the model to enable self-correction, adaptation, and dynamic decision-making.

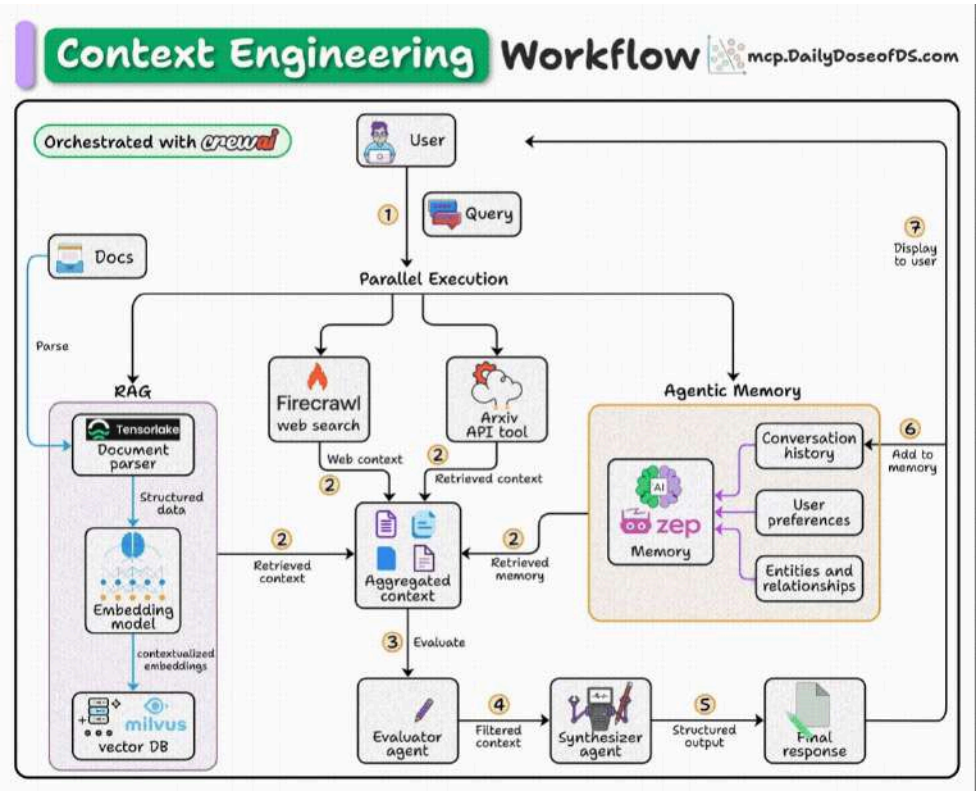
These are the exact six layers that help you build fully context-aware Agents.

Build a Context Engineering workflow

We'll build a multi-agent research assistant using context engineering principles.

This Agent will gather its context across 4 sources: Documents, Memory, Web search, and Arxiv.

Here's our workflow:

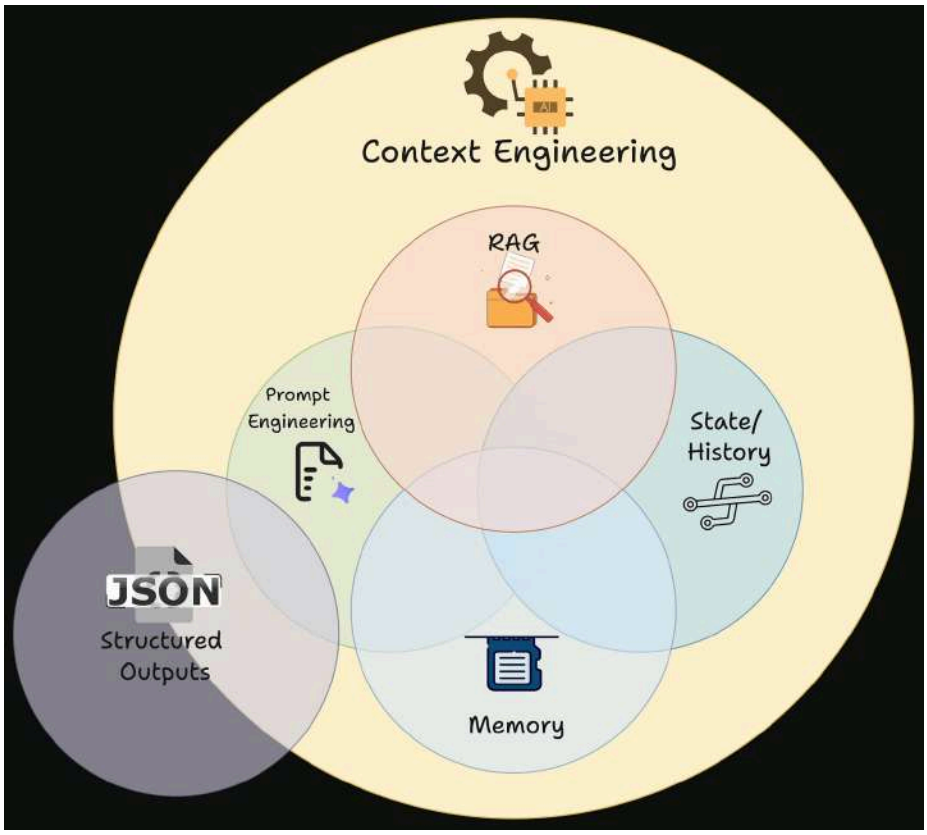


- User submits query.
- Fetch context from docs, web, arxiv API, and memory.
- Pass the aggregated context to an agent for filtering.
- Pass the filtered context to another agent to generate a response.
- Save the final response to memory.

Tech stack:

- Tensorlake to get RAG-ready data from complex docs
- Zep for memory
- Firecrawl for web search
- Milvus for vector DB
- CrewAI for orchestration

Let's go!



CE involves creating dynamic systems that offer:

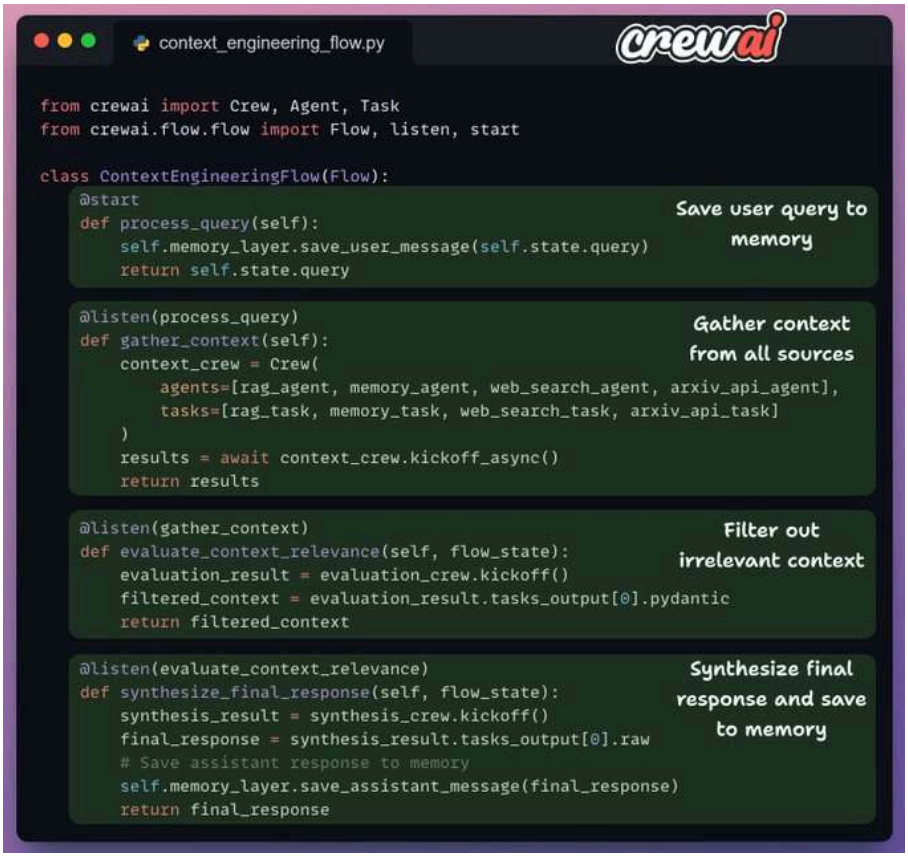
- The right info
- The right tools
- In the right format

This ensures the LLM can effectively complete the task.

#1) Crew flow

We'll follow a top-down approach to understand the code.

Here's an outline of what our flow looks like:



Note that this is one of many blueprints to implement a context engineering workflow. Your pipeline will likely vary based on the use case.

#2) Prepare data for RAG

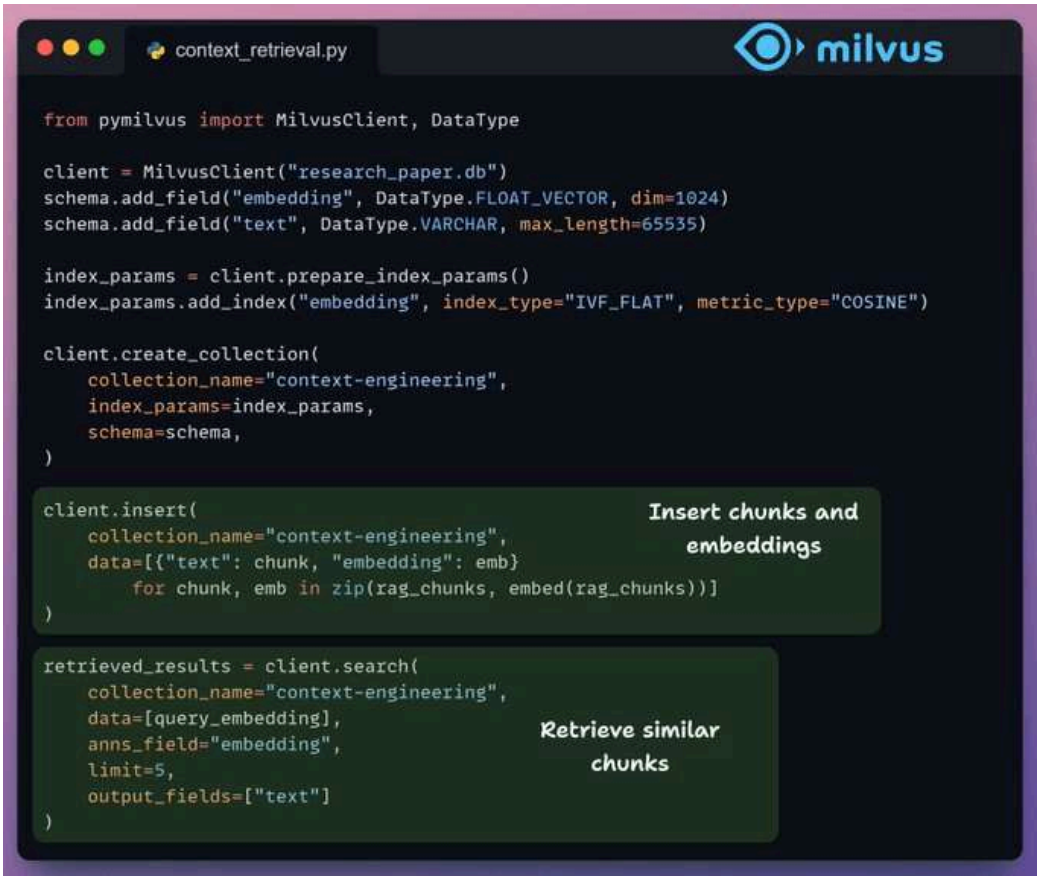
We use Tensorlake to convert the document into RAG-ready markdown chunks for each section.

The extracted data can be directly embedded and stored in a vector DB without further processing.

#3) Indexing and retrieval

Now that we have RAG-ready chunks along with the metadata, it's time to store them in a self-hosted Milvus vector database.

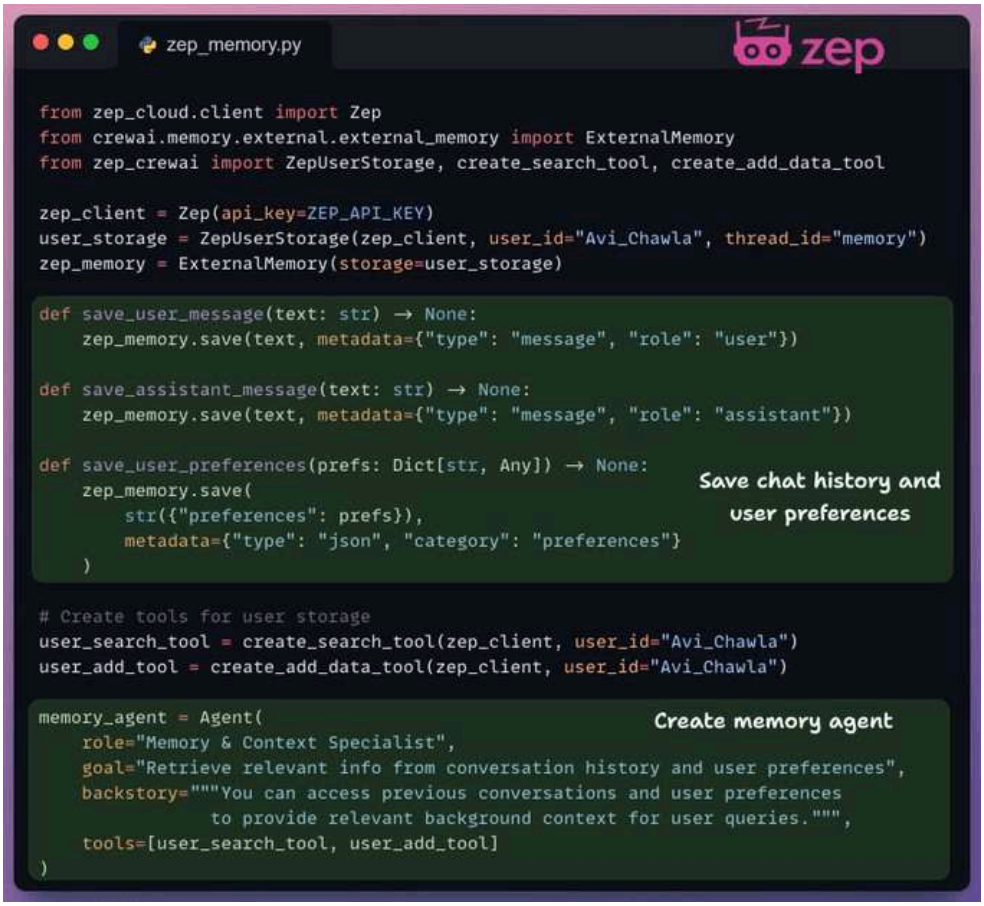
We retrieve the top-k most similar chunks to our query:



#4) Build memory layer

Zep acts as the core memory layer of our workflow. It creates temporal knowledge graphs to organize and retrieve context for each interaction.

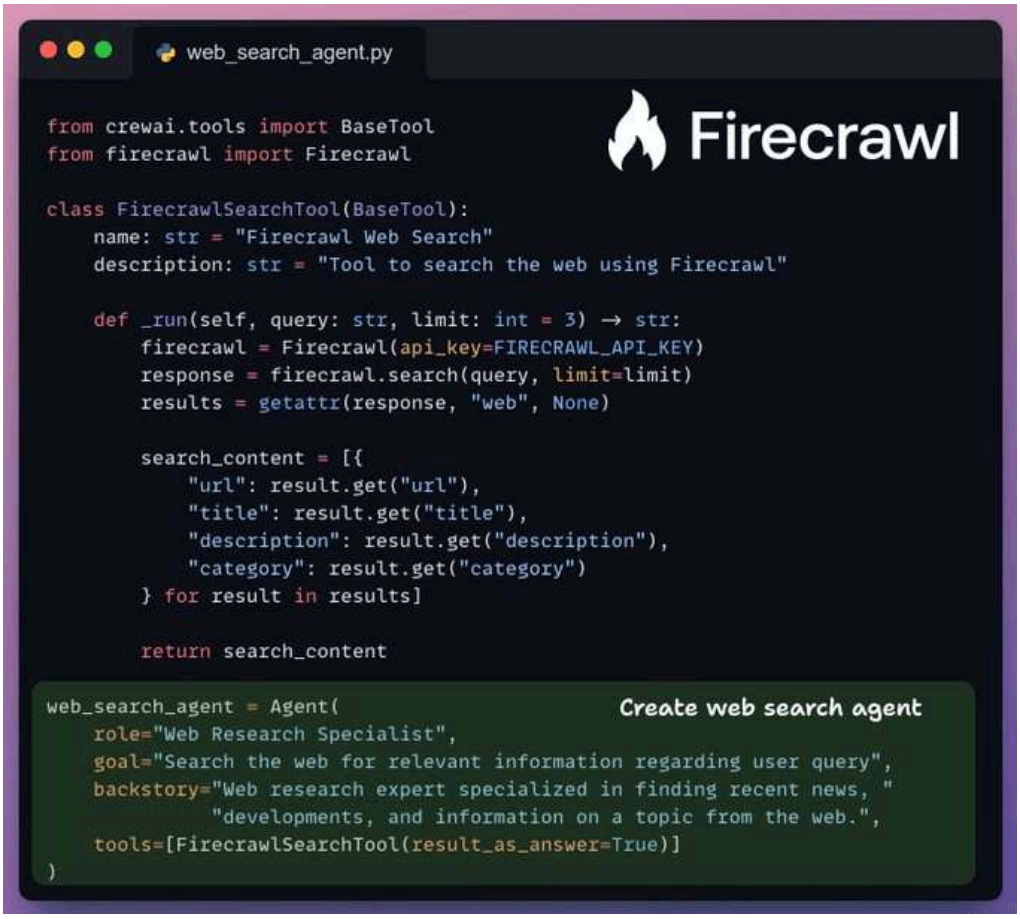
We use it to store and retrieve context from chat history and user data.



#5) Firecrawl web search

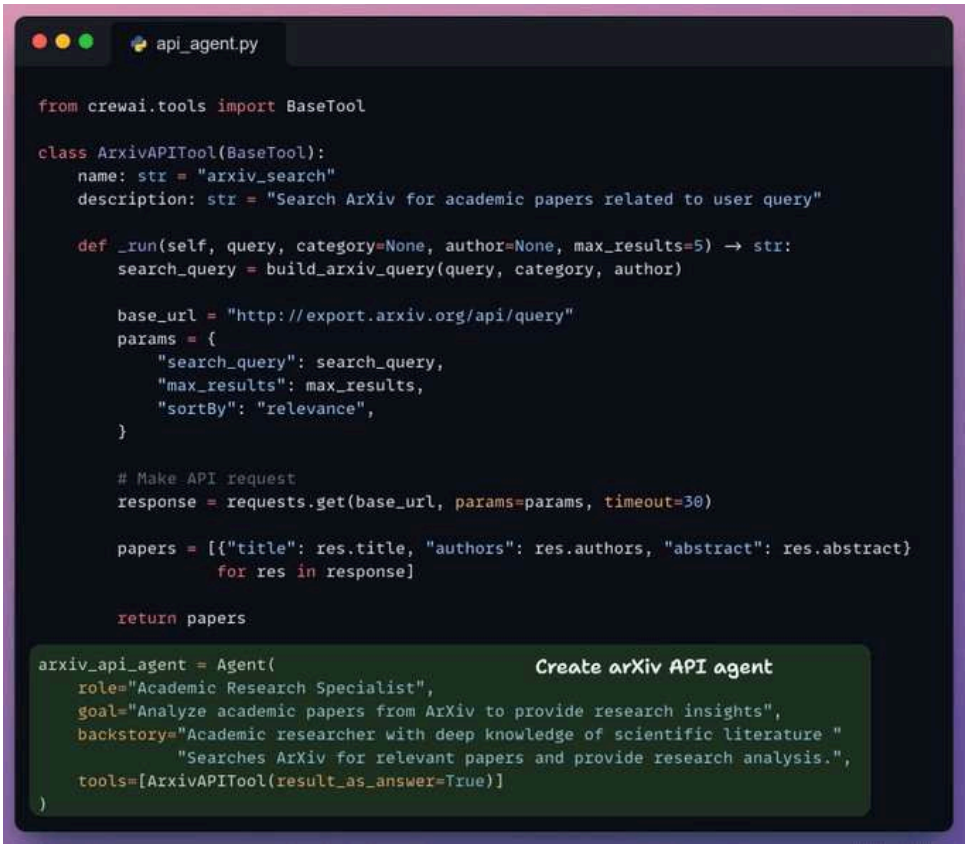
We use Firecrawl web search to fetch the latest news and developments related to the user query.

Firecrawl's v2 endpoint provides 10x faster scraping, semantic crawling, and image search, turning any website into LLM-ready data.



#6) ArXiv API search

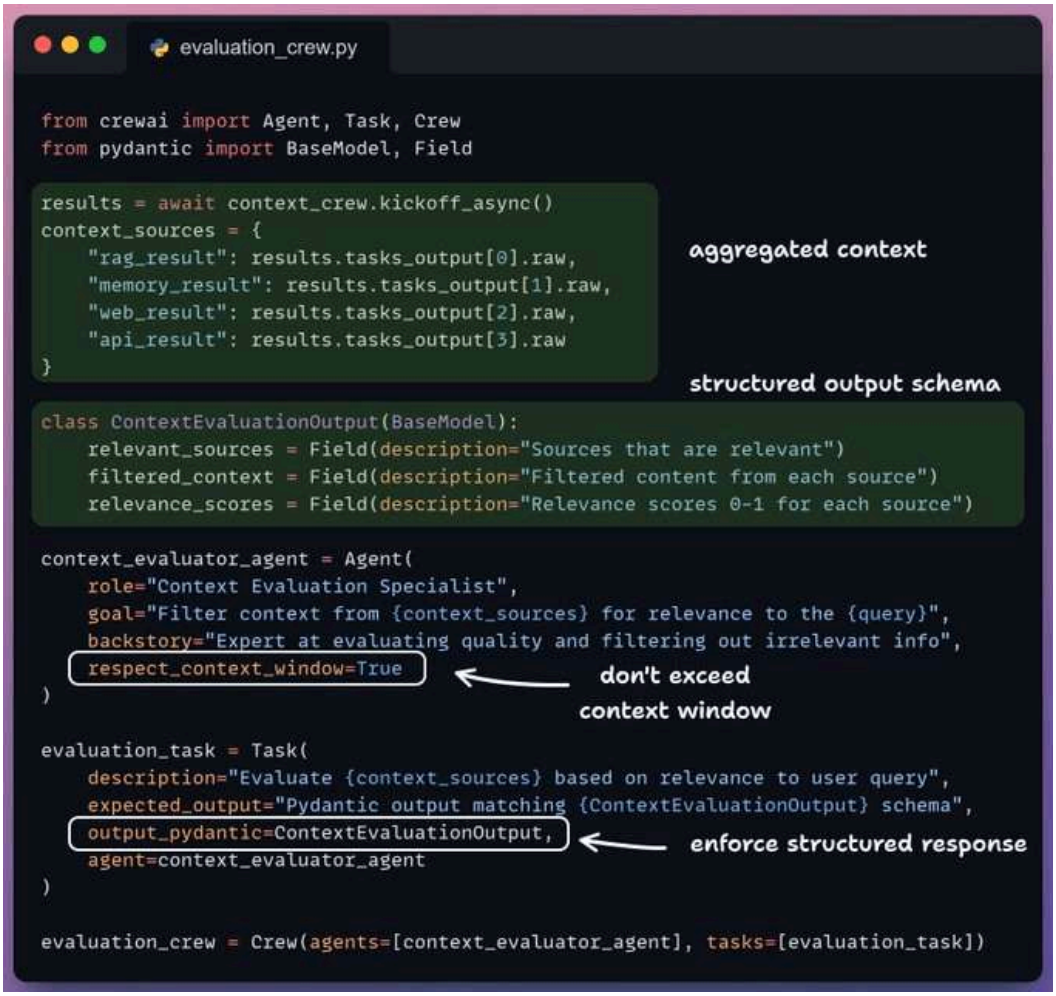
To further support research queries, we use the arXiv API to retrieve relevant results from their data repository based on the user query.



#7) Filter context

Now, we pass our combined context to the context evaluation agent that filters out irrelevant context.

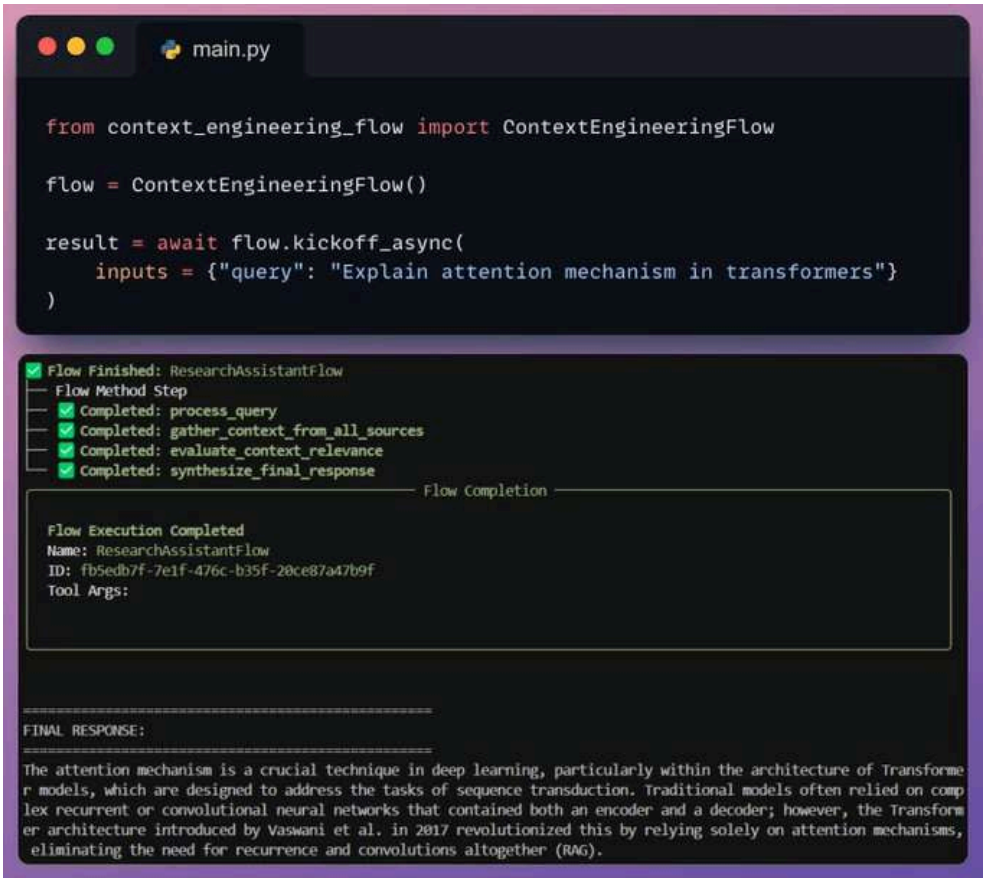
This filtered context is then passed to the synthesizer agent that generates the final response.



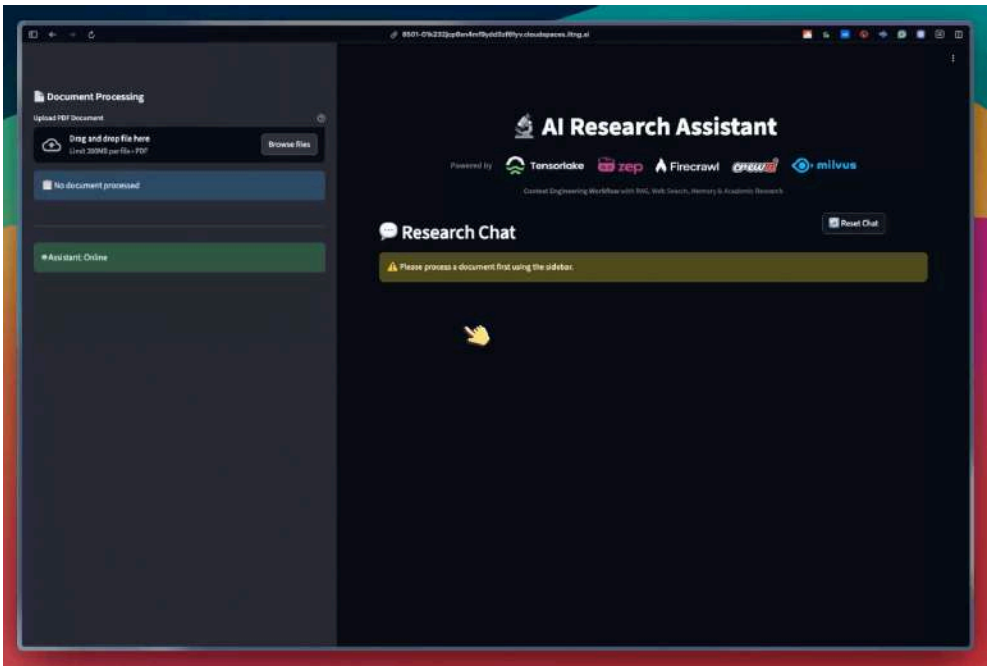
#8) Kick off the workflow

Finally, we kick off our context engineering workflow with a query.

Based on the query, we notice that the RAG tool, powered by Tensorlake, was the most relevant source for the LLM to generate a response.



We also translated this workflow into a streamlit app that:



- Provides citations with links and metadata.
- Provides insights into relevant sources.

The workflow explained above is one of the many blueprints. Your implementation can vary.

Context Engineering in Claude Skills

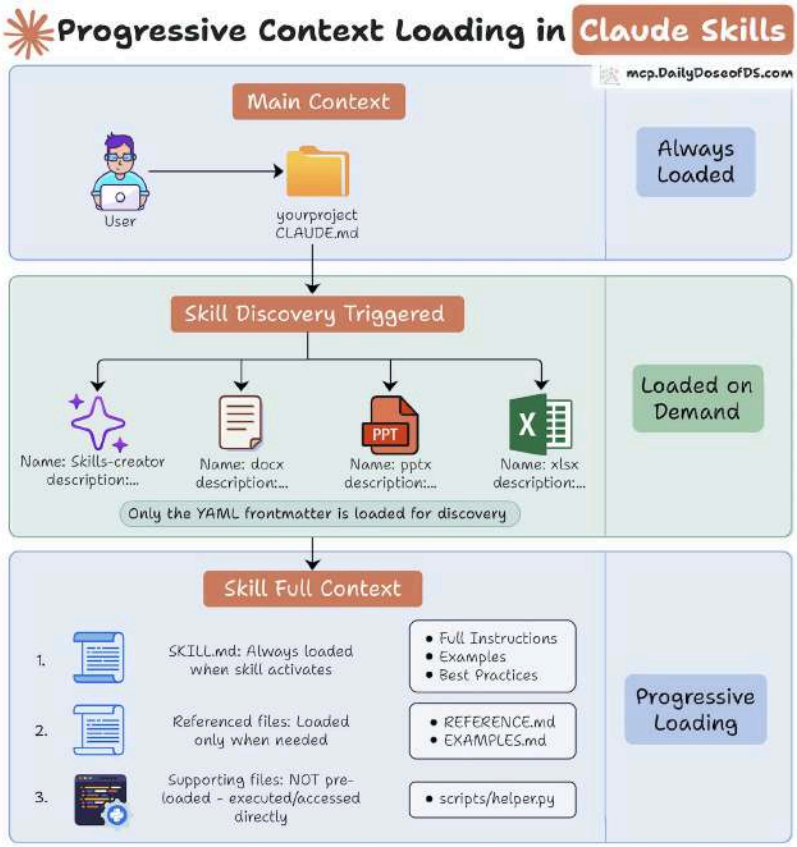
Claude Skills are Anthropic's mechanism for giving agents reusable, persistent abilities without overloading the model's context window.

They solve a practical issue in agent design: LLMs forget everything unless all instructions, examples and edge cases are restated each time.

Skills package this information into small, self-contained units that Claude loads only when they're relevant.

This allows an agent to use hundreds of specialized workflows while keeping its active context lightweight.

To make this scalable, Skills use a three-layer context management system that lets it use 100s of skills without hitting context limits.



Let's understand how it works:

- Layer 1: Main Context - Always loaded, it contains the project configuration.
- Layer 2: Skill Metadata - Comprises only the YAML frontmatter, about 2-3 lines (< 200 tokens).
- Layer 3: Active Skill Context - SKILL.md files and associated documentation are loaded as needed.

Supporting files like scripts and templates aren't pre-loaded but accessed directly when in use, consuming zero tokens.

This architecture supports hundreds of skills without breaching context limits.

Now let's zoom into the main ideas behind Skills, because understanding what they are clarifies why this 3-layer system matters.

Skills as SOPs for Agents

Think of a Skill as a packaged procedure - a complete, reusable workflow that teaches the agent how to perform a task with consistency.

Instead of re-explaining steps, examples, constraints, and edge cases every time, you define the workflow once and reuse it forever.

It's the AI equivalent of an operating manual: structured, repeatable, and self-contained.

Anatomy of a Skill

A skill is simply a folder, but what it contains is carefully designed:

- A skill.md file with two layers of context:
  - YAML Front Matter: a tiny descriptor Claude uses to decide when the skill is relevant.
  - Skill Body: the detailed instructions, workflows, examples, and guidance used during execution.
- Optional supporting files such as scripts, templates or reference docs. These aren't loaded into context, they're fetched only when the agent needs them.

This separation lets Claude stay lightweight until a specific skill is activated.

How Skills Fit Into the Agent Architecture

Skills don't replace Projects, Subagents or MCP - they complement them:

- Projects organize your workspace.
- MCP connects Claude to tools and external services.
- Subagents handle delegated reasoning.
- Skills package the reusable expertise that all of them can rely on.

Each solves a different layer of the agent problem, and skills serve as the procedural knowledge base.

Building Your Own Skills

The creation process is straightforward:

1. Identify a workflow you repeat constantly.
2. Create a skill folder and add a skill.md file.
3. Write the YAML front matter + full markdown instructions.
4. Add any scripts, examples, or supporting resources.
5. Zip the folder and upload it in Claude's capabilities.

Claude Desktop even includes a "Skill Creator" skill that helps generate the structure for you.

Manual RAG Pipeline vs Agentic Context Engineering

Imagine you have data that's spread across several sources (Gmail, Drive, etc.).

How would you build a unified query engine over it?

Devs would typically treat context retrieval like a weekend project.

...and their approach would be: "Embed the data, store in a vector DB and do RAG."

This works beautifully for static sources.

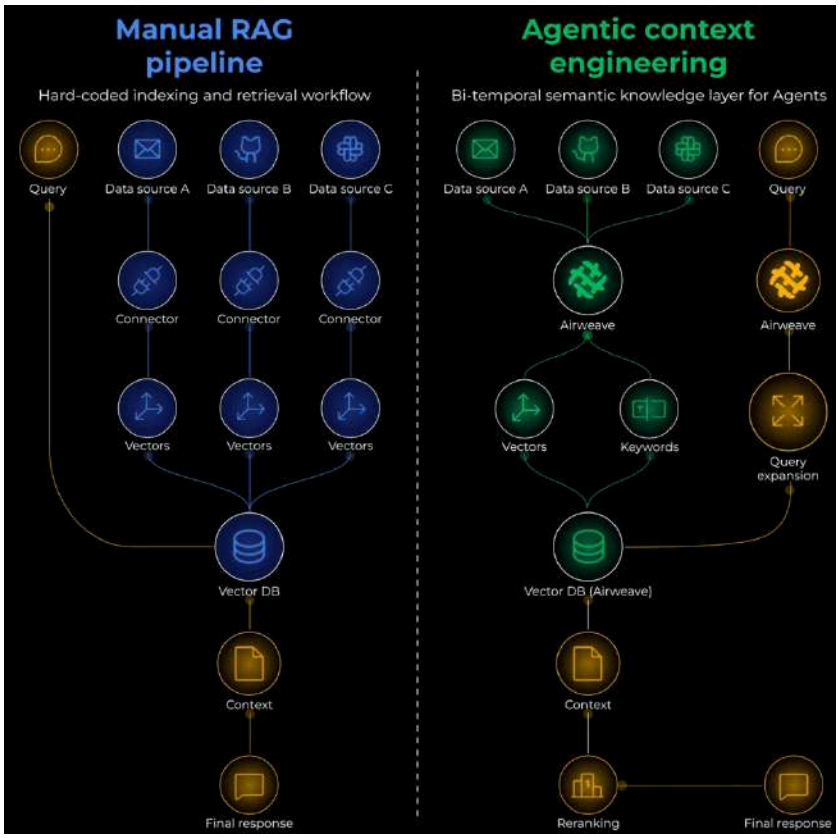
But the problem is that no real-world workflow looks like this.

To understand better, consider this query:

What's blocking the Chicago office project, and when's our next meeting about it?

Answering this single query requires searching across sources like Linear (for blockers), Calendar (for meetings), Gmail (for emails), and Slack (for discussions).

No naive RAG setup can handle this!



To actually solve this problem, you'd need to think of it as building an Agentic context retrieval system with three critical layers:

- Ingestion layer:
    - Connect to apps without auth headaches.
    - Process different data sources properly before embedding (email vs code vs calendar).
    - Detect if a source is updated and refresh embeddings (ideally, without a full refresh).

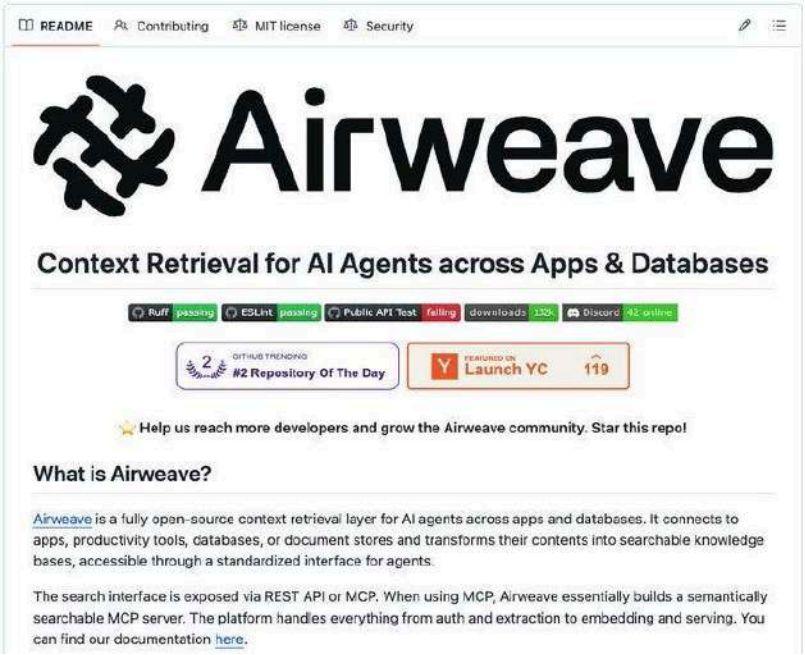
- Retrieval layer:
    - Expand vague queries to infer what users actually want.
    - Direct queries to the correct data sources.
    - Layer multiple search strategies like semantic-based, keyword-based, and graph-based.
    - Ensure retrieving only what users are authorized to see.
    - Weigh old vs. new retrieved info (recent data matters more, but old context still counts).
- Generation layer:
    - Provide a citation-backed LLM response.

That's months of engineering before your first query works.

It's definitely a tough problem to solve...

...but this is precisely how giants like Google (in Vertex AI Search), Microsoft (in M365 products), AWS (in Amazon Q Business), etc., are solving it.

If you want to see it in practice, this approach is actually implemented in Airweave, a recently trending 100% open-source framework that provides the context retrieval layer for AI agents across 30+ apps and databases(as of 3 Dec,2025).



It implements everything we discussed above, like:

- How to handle authentication across apps.
- How to process different data sources.
- How to gather info from multiple tools.
- How to weigh old vs. new info.
- How to detect updates and do real-time sync.
- How to generate perplexity-like citation-backed responses, and more.

For instance, to detect updates and initiate a re-sync, one might do timestamp comparisons.

But this does not tell if the content actually changed (maybe only the permission was updated), and you might still re-embed everything unnecessarily.

Airweave handles this by implementing source-specific hashing techniques like entity-level hashing, file content hashing, cursor-based syncing, etc.

You can see the full implementation on GitHub and try it yourself.

But the core insight applies regardless of the framework you use:

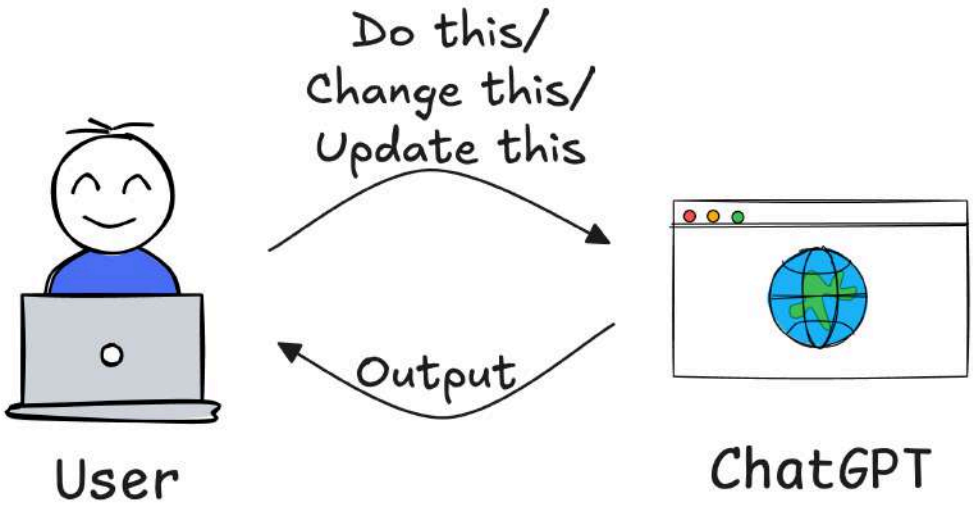
Context retrieval for Agents is an infrastructure problem, not an embedding problem.

You need to build for continuous sync, intelligent chunking, and hybrid search from day one.

AI Agents

What is an AI Agent?

Imagine you want to generate a report on the latest trends in AI research. If you use a standard LLM, you might:

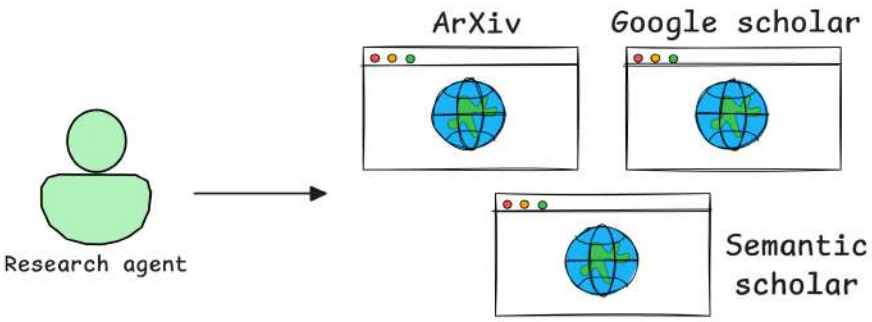


1. Ask for a summary of recent AI research papers.
2. Review the response and realize you need sources.
3. Obtain a list of papers along with citations.
4. Find that some sources are outdated, so you refine your query.
5. Finally, after multiple iterations, you get a useful output.

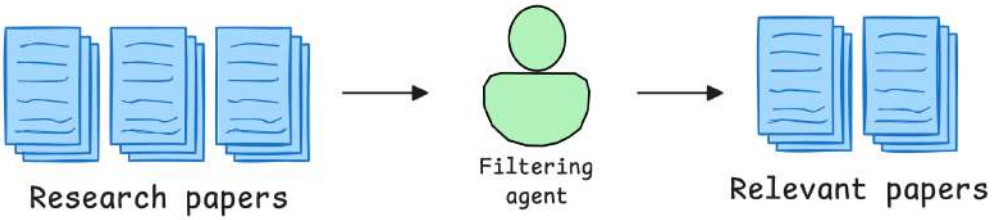
This iterative process takes time and effort, requiring you to act as the decision-maker at every step.

Now, let's see how AI agents handle this differently:

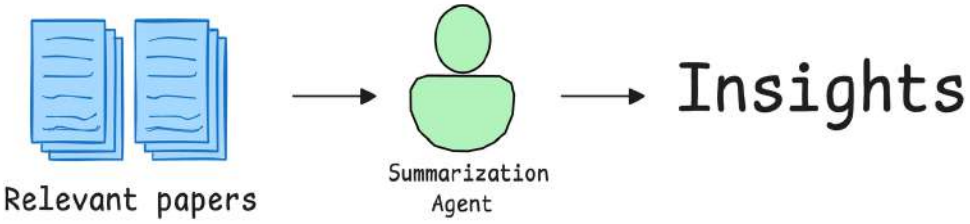
A Research Agent autonomously searches and retrieves relevant AI research papers from arXiv, Semantic Scholar, or Google Scholar.



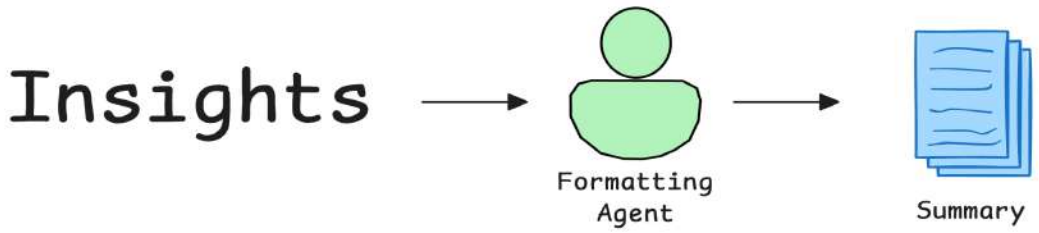
- A Filtering Agent scans the retrieved papers, identifying the most relevant ones based on citation count, publication date, and keywords.



- A Summarization Agent extracts key insights and condenses them into an easy-to-read report.



- A Formatting Agent structures the final report, ensuring it follows a clear, professional layout.



Here, the AI agents not only execute the research process end-to-end but also self-refine their outputs, ensuring the final report is comprehensive, up-to-date, and well-structured - all without requiring human intervention at every step.

To formalize AI Agents are autonomous systems that can reason, think, plan, figure out the relevant sources and extract information from them when needed, take actions, and even correct themselves if something goes wrong.

Agent vs LLM vs RAG

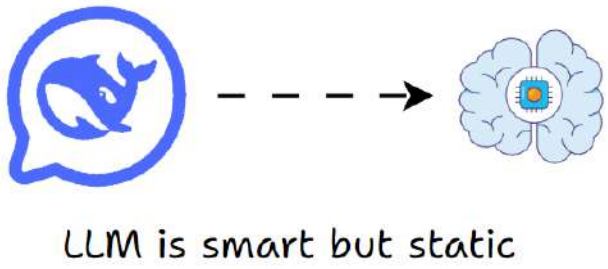
Let's break it down with a simple analogy:

- LLM is the brain.
- RAG is feeding that brain with fresh information.
- An agent is the decision-maker that plans and acts using the brain and the tools.

LLM (Large Language Model)

An LLM like GPT-4 is trained on massive text data.

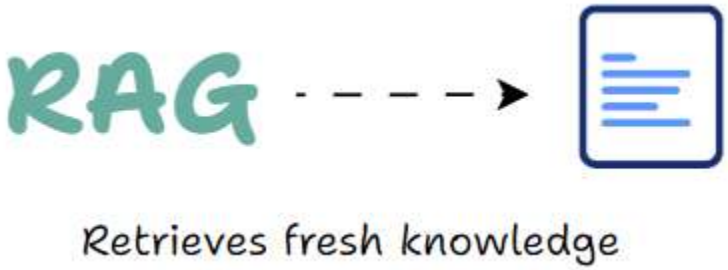
It can reason, generate, summarize but only using what it already knows (i.e., its training data).



It's smart, but static. It can't access the web, call APIs, or fetch new facts on its own.

RAG (Retrieval-Augmented Generation)

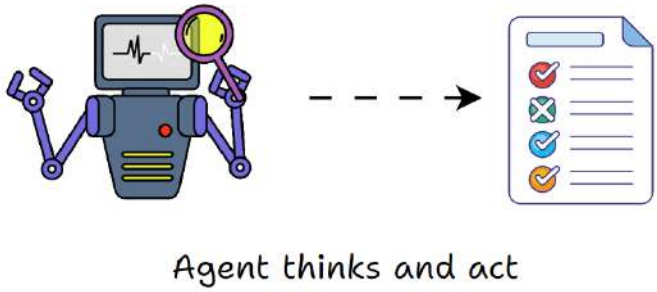
RAG enhances an LLM by retrieving external documents (from a vector DB, search engine, etc.) and feeding them into the LLM as context before generating a response.



RAG makes the LLM aware of updated, relevant info without retraining.

Agent

An Agent adds autonomy to the mix.



It doesn't just answer a question—it decides what steps to take:

Should it call a tool? Search the web? Summarize? Store info?

An Agent uses an LLM, calls tools, makes decisions, and orchestrates workflows just like a real assistant.

Building blocks of AI Agents

AI agents are designed to reason, plan, and take action autonomously. However,

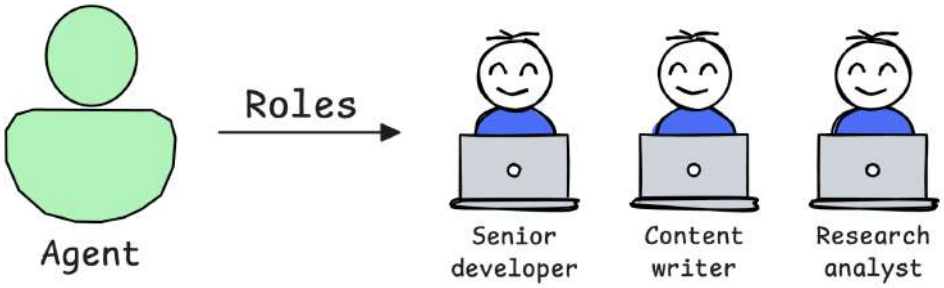
to be effective, they must be built with certain key principles in mind. There are six essential building blocks that make AI agents more reliable, intelligent, and useful in real-world applications:

1. Role-playing
2. Focus
3. Tools
4. Cooperation
5. Guardrails
6. Memory

Let's explore each of these concepts and understand why they are fundamental to building great AI agents.

1) Role-playing

One of the simplest ways to boost an agent's performance is by giving it a clear, specific role.



A generic AI assistant may give vague answers. But define it as a "Senior contract lawyer," and it responds with legal precision and context.

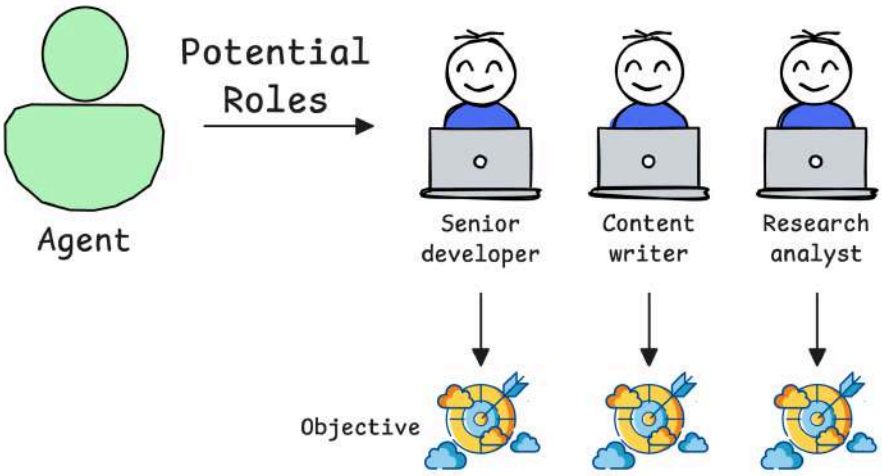
Why?

Because role assignment shapes the agent's reasoning and retrieval process. The more specific the role, the sharper and more relevant the output.

2) Focus/Tasks

Focus is key to reducing hallucinations and improving accuracy.

Giving an agent too many tasks or too much data doesn't help - it hurts.



Overloading leads to confusion, inconsistency, and poor results.

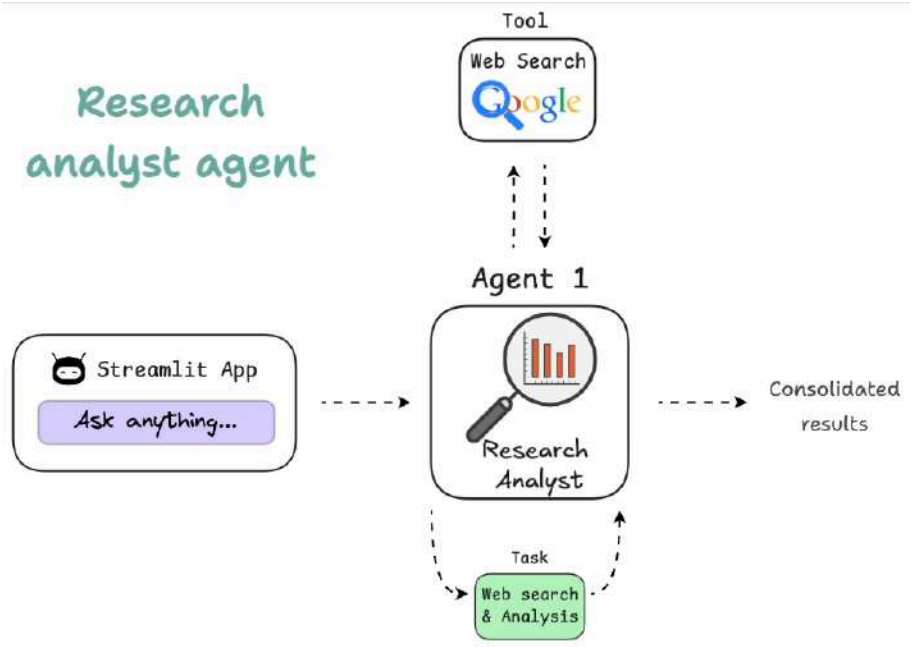
For example, a marketing agent should stick to messaging, tone, and audience not pricing or market analysis.

Instead of trying to make one agent do everything, a better approach is to use multiple agents, each with a specific and narrow focus.
Specialized agents perform better - every time.

3) Tools

Agents get smarter when they can use the right tools.

But more tools ≠ better results.



For example, an AI research agent could benefit from:

- A web search tool for retrieving recent publications.
- A summarization model for condensing long research papers.
- A citation manager to properly format references.

But if you add unnecessary tools—like a speech-to-text module or a code execution environment—it could confuse the agent and reduce efficiency.

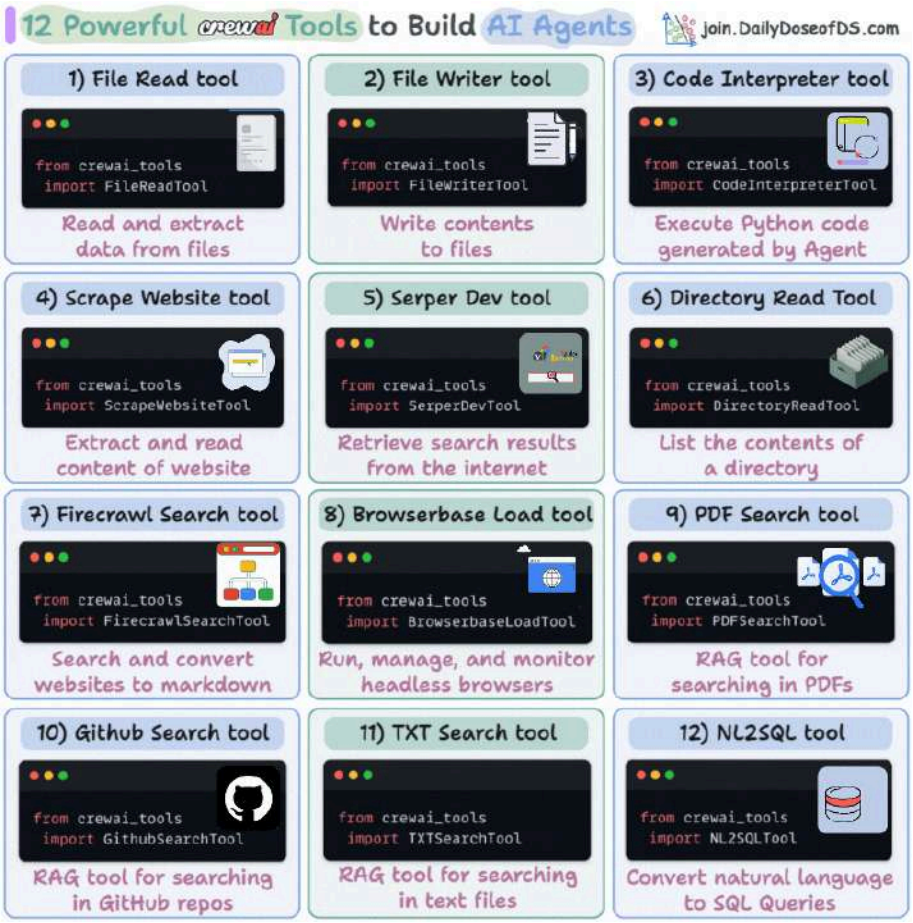
#3.1) Custom tools

While LLM-powered agents are great at reasoning and generating responses, they lack direct access to real-time information, external systems, and specialized computations.

Tools allow the Agent to:

- Search the web for real-time data.
- Retrieve structured information from APIs and databases.
- Execute code to perform calculations or data transformations.
- Analyze images, PDFs, and documents beyond just text inputs.

CrewAI supports several tools that you can integrate with Agents, as depicted below:

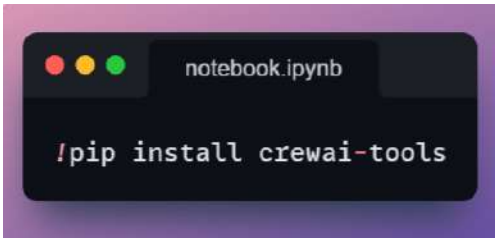


However, you may need to build custom tools at times.

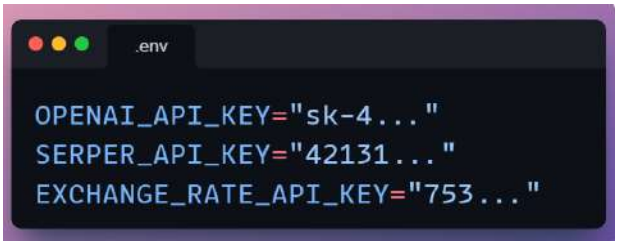
In this example, we're building a real-time currency conversion tool inside CrewAI. Instead of making an LLM guess exchange rates, we integrate a custom tool that fetches live exchange rates from an external API and provides some insights.

Below, let's look at how you can build one for your custom needs in the CrewAI framework.

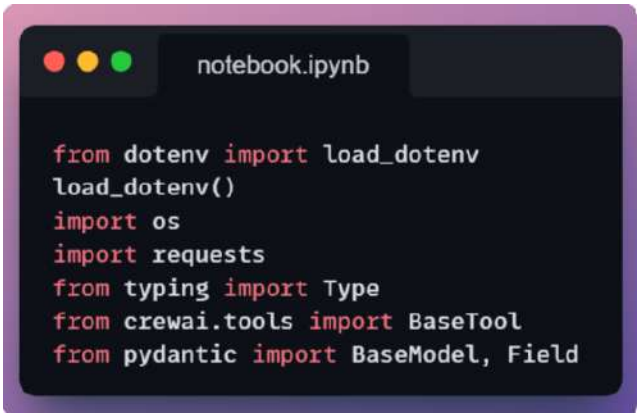
Firstly, make sure the tools package is installed:



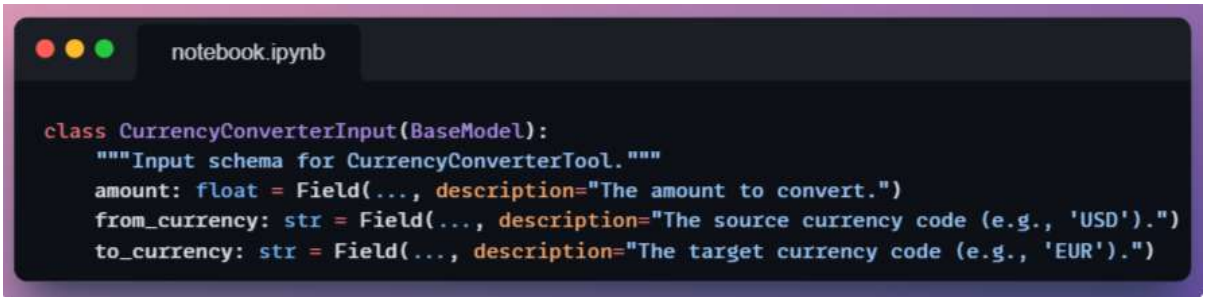
You would also need an API key from here: https://www.exchangerate-api.com/ (it's free). Specify it in the .env file as shown below:



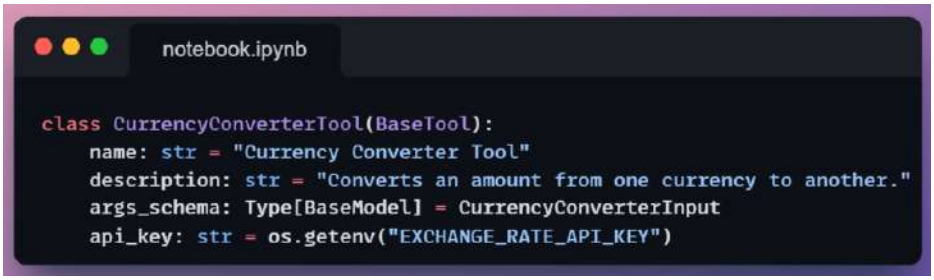
Once that's done, we start with some standard import statements:



Next, we define the input fields the tool expects using Pydantic.

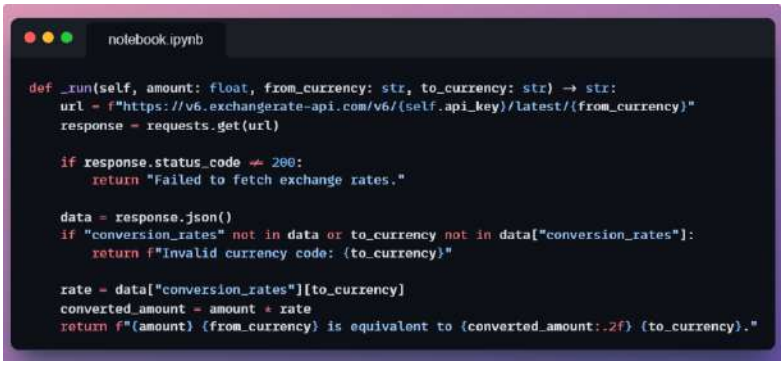


Now, we define the CurrencyConverterTool by inheriting from *BaseTool:*



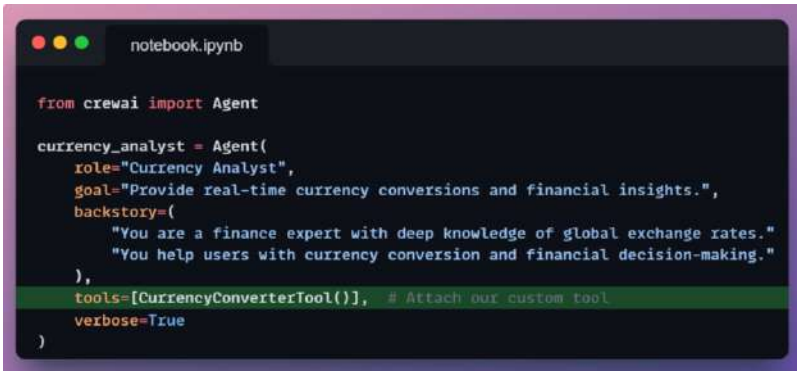
Every tool class should have the _run method which we will execute whenever the Agents wants to make use of it.

For our use case, we implement it as follows:

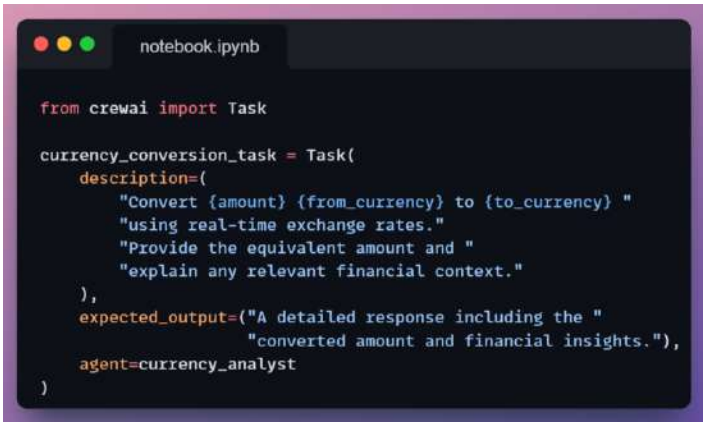


In the above code, we fetch live exchange rates using an API request. We also handle errors if the request fails or the currency code is invalid.

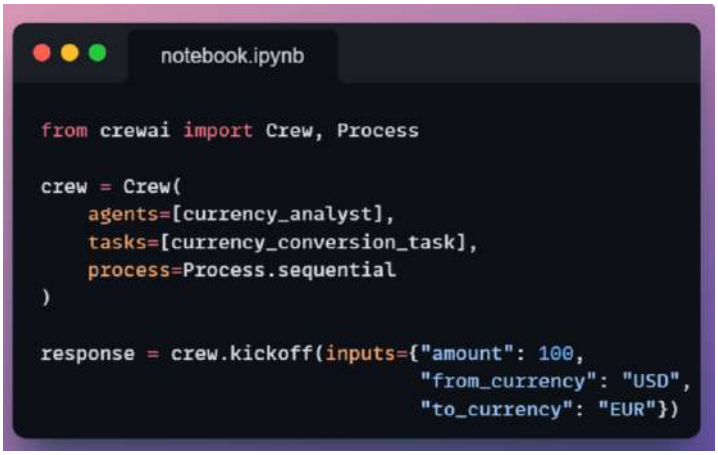
Now, we define an agent that uses the tool for real-time currency analysis and attach our CurrencyConverterTool, allowing the agent to call it directly if needed:



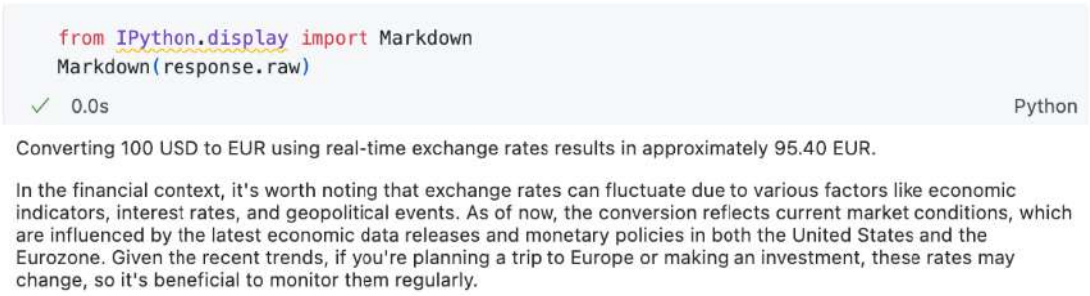
We assign a task to the currency_analyst agent.



Finally, we create a Crew, assign the agent to the task, and execute it.



Printing the response, we get the following output:



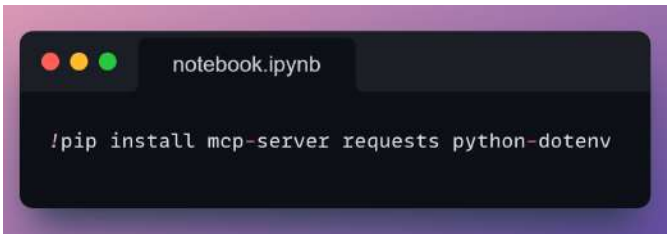
Works as expected!

#3.2) Custom tools via MCP

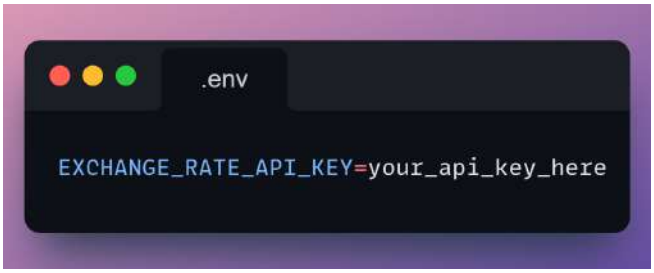
Now, let's take it a step further.

Instead of embedding the tool directly in every Crew, we'll expose it as a reusable MCP tool - making it accessible across multiple agents and flows via a simple server.

First, install the required packages:



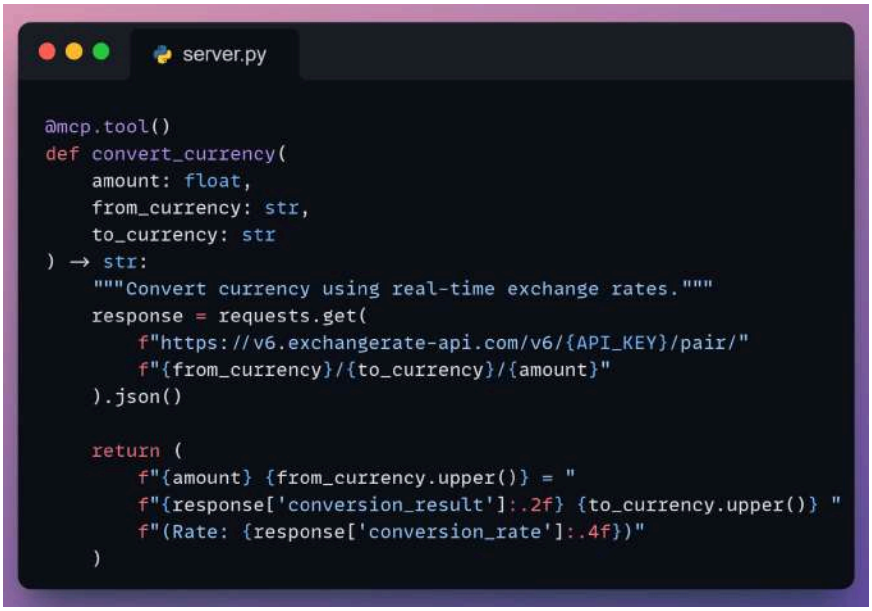
We'll continue using ExchangeRate-API in our .env file:



We'll now write a lightweight server.py script that exposes the currency converter tool. We start with the standard imports:

Now, we load environment variables and initialize the server:

Next, we define the tool logic with @mcp.tool():



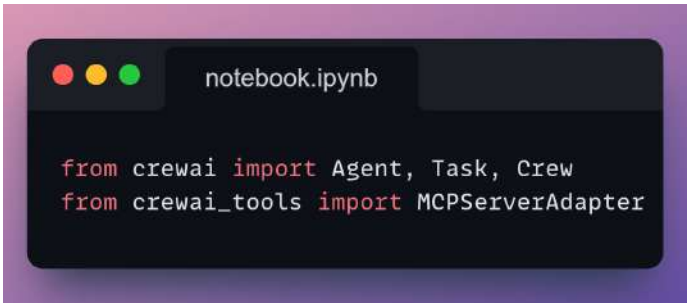
This function takes three inputs - amount, source currency, and target currency and returns the converted result using the real-time exchange rate API.

To make the tool accessible, we need to run the MCP server. Add this at the end of your script:

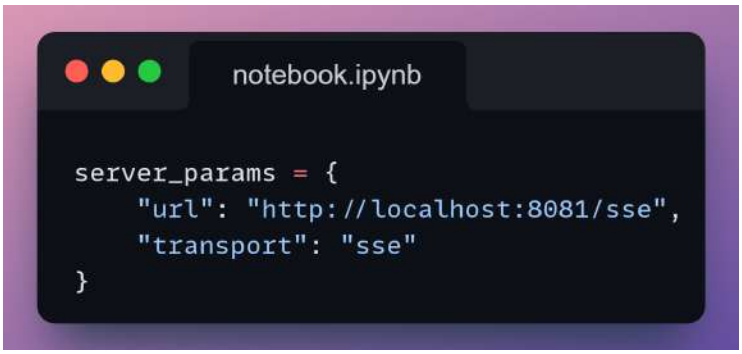
This starts the server and exposes your convert_currency tool at: http://localhost:8081/sse.

Now any CrewAI agent can connect to it using MCPServerAdapter. Let's now consume this tool from within a CrewAI agent.

First, we import the required CrewAI classes. We'll use Agent, Task, and Crew from CrewAI, and MCPServerAdapter to connect to our tool server.

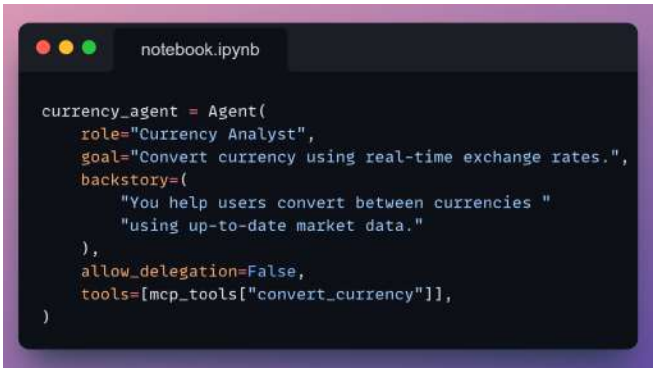


Next, we connect to the MCP tool server. Define the server parameters to connect to your running tool (from server.py).



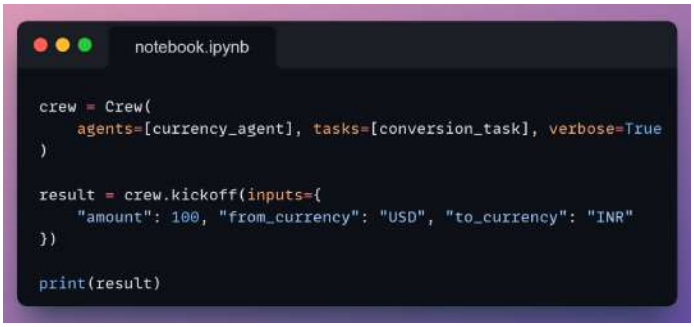
Now, we use the discovered MCP tool in an agent:

This agent is assigned the convert_currency tool from the remote server. It can now call the tool just like a locally defined one.

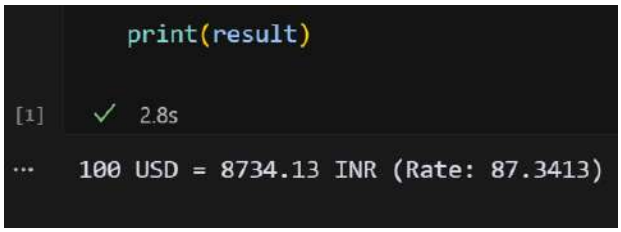


We give the agent a task description:

Finally, we create the Crew, pass in the inputs and run it:



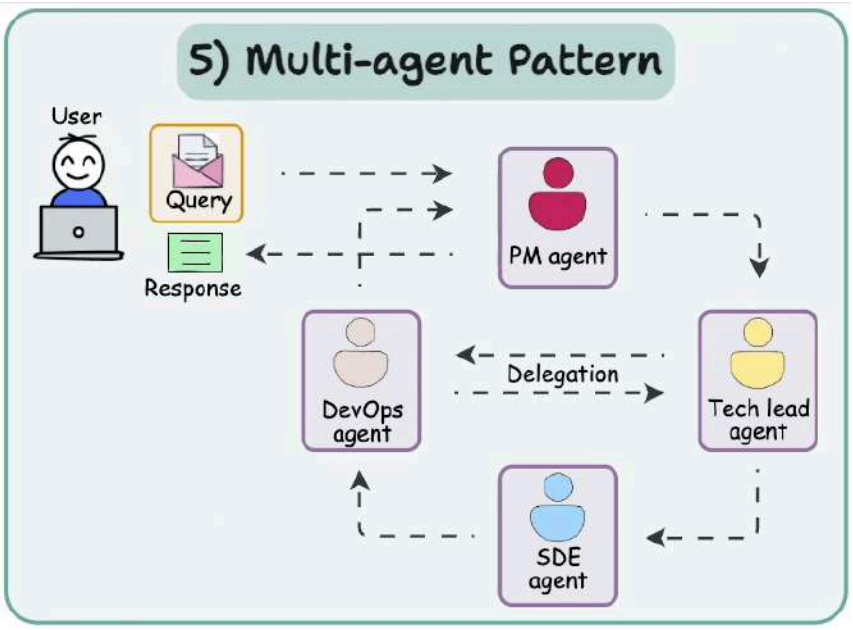
Printing the result, we get the following output:



4) Cooperation

Multi-agent systems work best when agents collaborate and exchange feedback.

Instead of one agent doing everything, a team of specialized agents can split tasks and improve each other's outputs.



Consider an AI-powered financial analysis system:

- One agent gathers data
- another assesses risk,
- a third builds strategy,
- and a fourth writes the report

Collaboration leads to smarter, more accurate results.

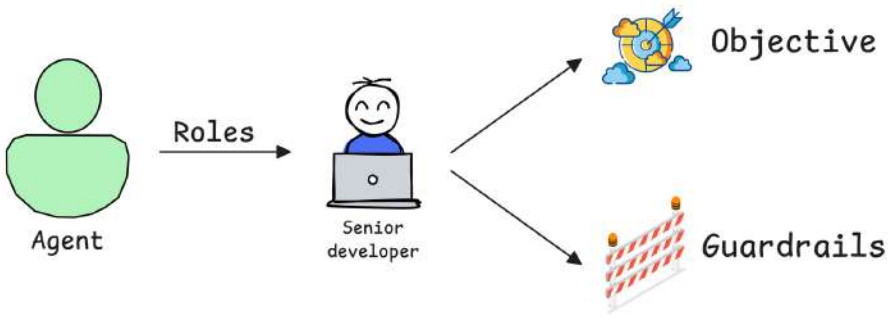
The best practice is to enable agent collaboration by designing workflows where agents can exchange insights and refine their responses together.

5) Guardrails

Agents are powerful but without constraints, they can go off track. They might

hallucinate, loop endlessly, or make bad calls.

Guardrails ensure that agents stay on track and maintain quality standards.



Examples of useful guardrails include:

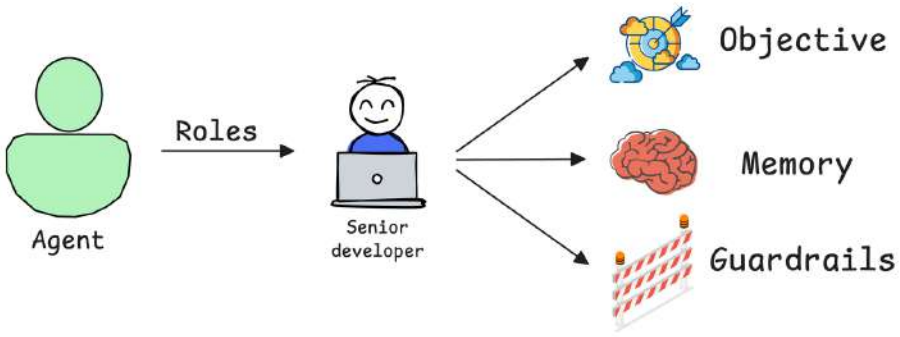
- Limiting tool usage: Prevent an agent from overusing APIs or generating irrelevant queries.
- Setting validation checkpoints: Ensure outputs meet predefined criteria before moving to the next step.
- Establishing fallback mechanisms: If an agent fails to complete a task, another agent or human reviewer can intervene.

For example, an AI-powered legal assistant should avoid outdated laws or false claims - guardrails ensure that.

6) Memory

Finally, we have memory, which is one of the most critical components of AI agents.

Without memory, an agent would start fresh every time, losing all context from previous interactions. With memory, agents can improve over time, remember past actions, and create more cohesive responses.



Different types of memory in AI agents include:

- Short-term memory – Exists only during execution (e.g., recalling recent conversation history).
- Long-term memory – Persists after execution (e.g., remembering user preferences over multiple interactions).
- Entity memory – Stores information about key subjects discussed (e.g., tracking customer details in a CRM agent).

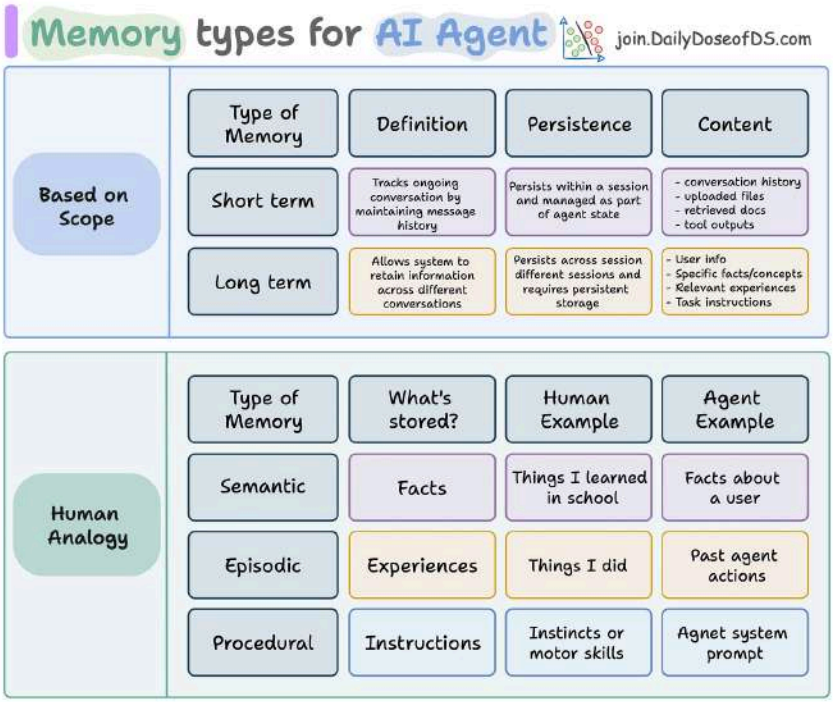
For example, in an AI-powered tutoring system, memory allows the agent to recall past lessons, tailor feedback, and avoid repetition.

Memory Types in AI Agents

Now, let us look at the memory types for AI agents in more detail.

Just like humans, long-term memory in agents can be:

- Semantic → Stores facts and knowledge
- Episodic → Recalls past experiences or task completions
- Procedural → Learns how to do things (think: internalized prompts/instructions)



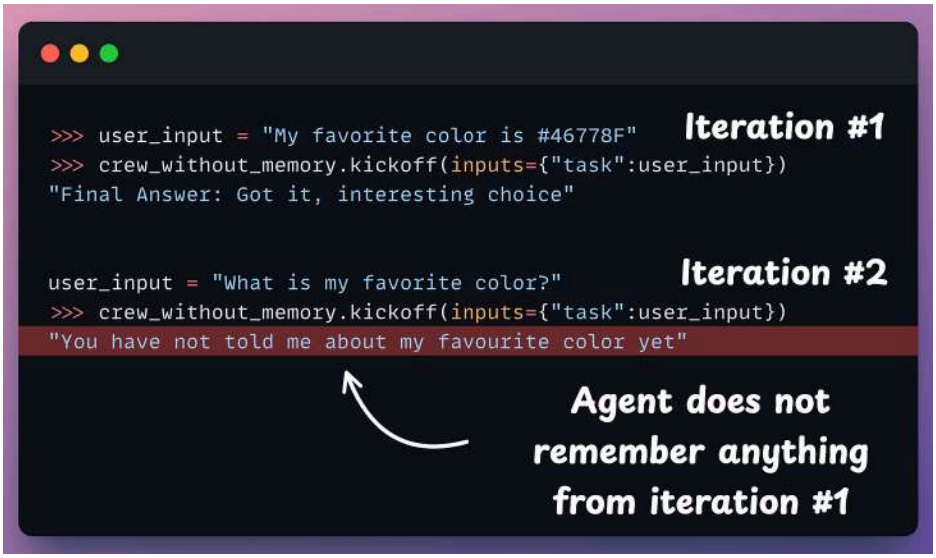
This memory isn't just nice-to-have but it enables agents to learn from past interactions without retraining the model.

This is especially powerful for continual learning: letting agents adapt to new tasks without touching LLM weights.

Importance of Memory for Agentic Systems

Let us now understand why memory is so powerful for Agentic systems?

Consider an Agentic system without Memory (below):



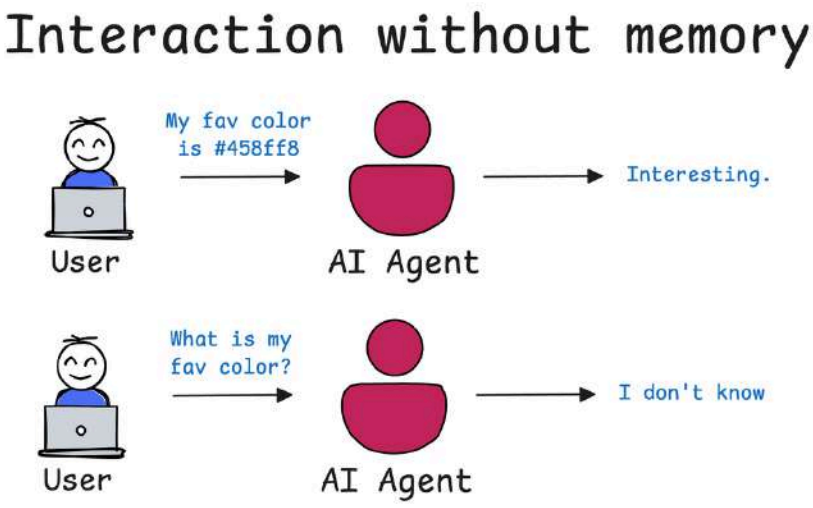
- In iteration #1, the user mentions their favorite color.
- In iteration #2, the Agent knows nothing about iteration #1.

This means the Agent is mostly stateless, and it has no recall abilities.

But now consider an Agentic system built with Memory (below):

- In iteration #1, the user mentions their favorite color.
- In iteration #2, the Agent can recall iteration #1.

Memory matters because if a memory-less Agentic system is deployed in production, every interaction with that Agent will be a blank slate.

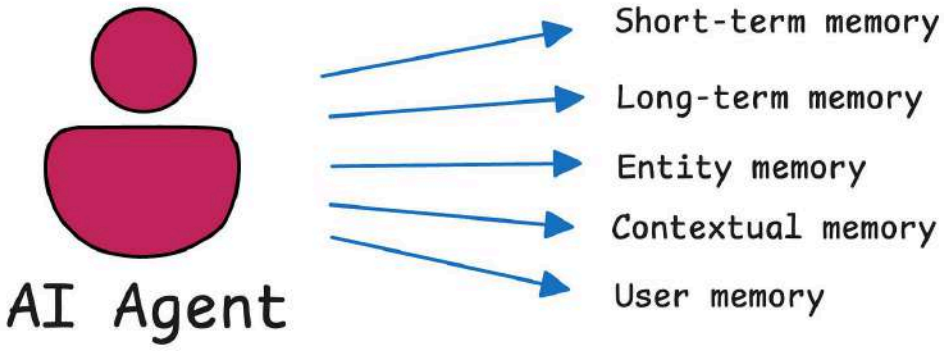


It doesn't matter if the user told the Agent their name five seconds ago, it's forgotten. If the Agent helped troubleshoot an issue in the last session, it won't remember any of it now.

With Memory, your Agent becomes context-aware and practically applicable.

But Memory isn't an abstract concept.

If you dive deeper, it follows a structured and intuitive architecture with several types of Memory.

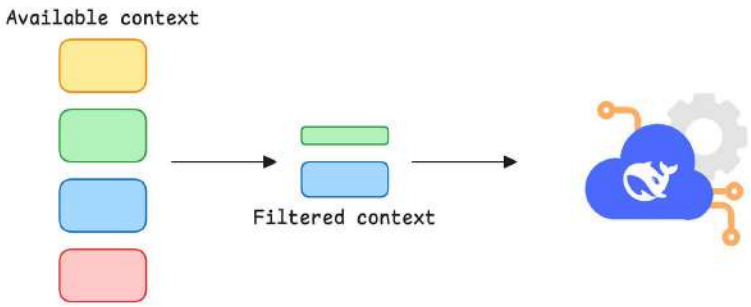


- Short-Term Memory
- Long-Term Memory
- Entity Memory
- Contextual Memory, and

- User Memory

Each serves a unique purpose in helping agents "remember" and utilize past information.

To simulate memory, the system has to manage context explicitly: choosing what to keep, what to discard, and what to retrieve before each new model call.

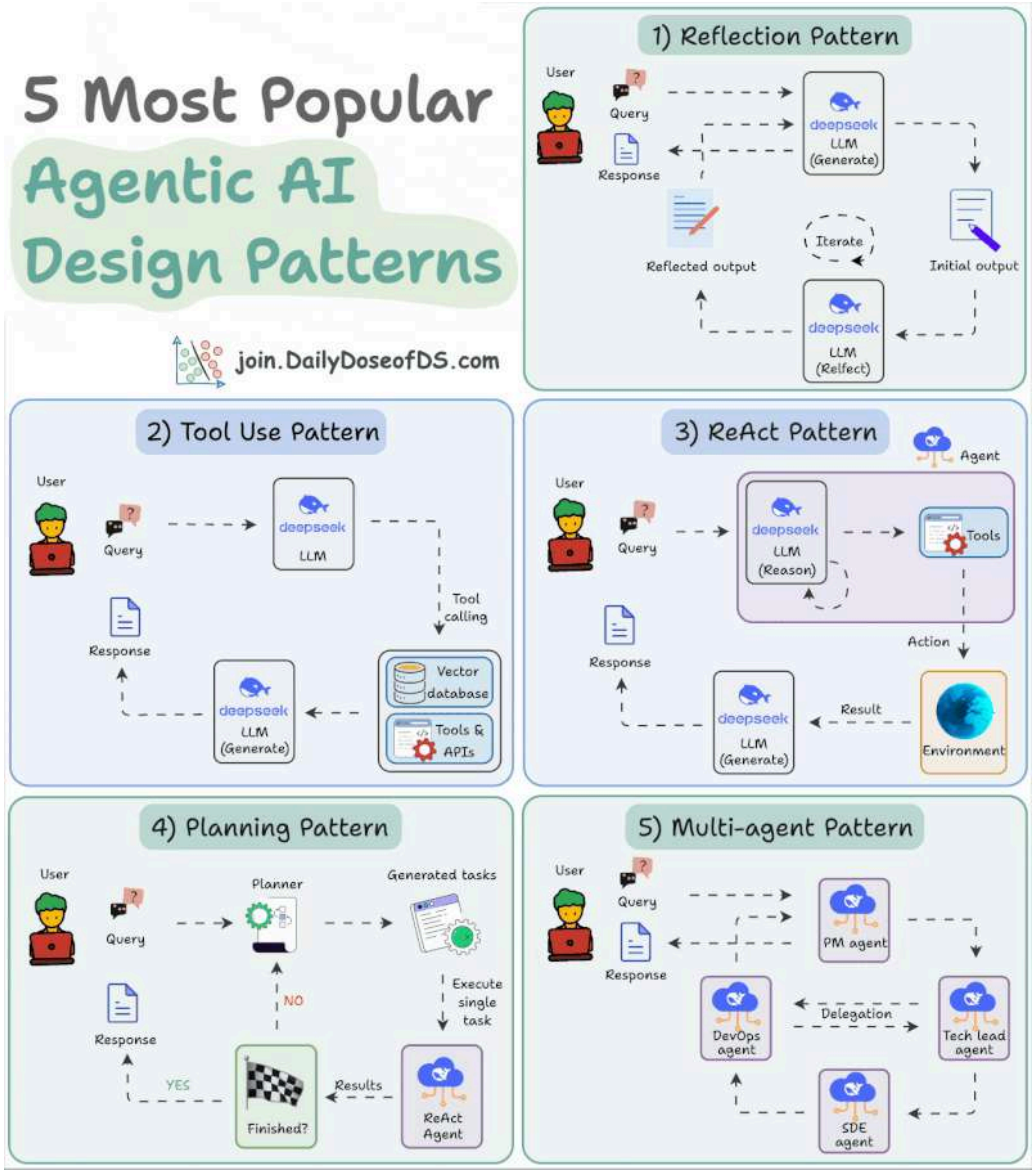


This is why memory is not a property of the model itself. It is a system design problem.

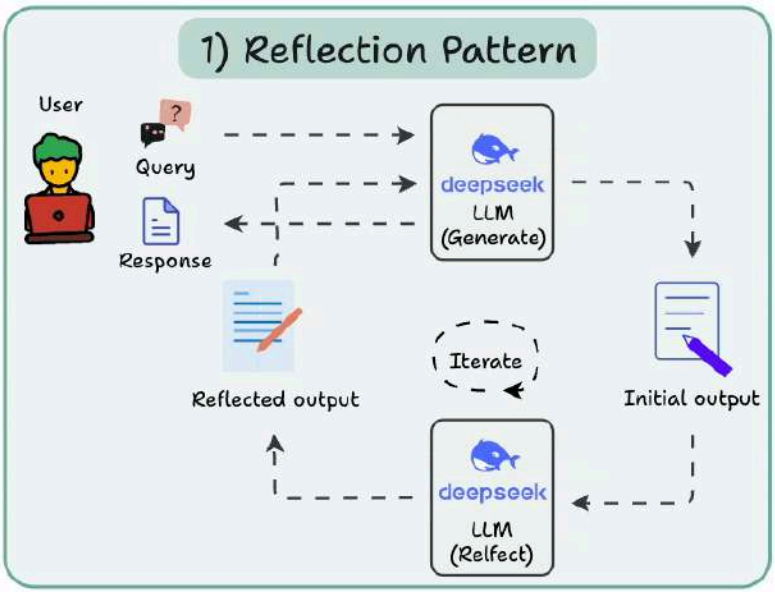
5 Agentic AI Design Patterns

Agentic behaviors allow LLMs to refine their output by incorporating self-evaluation, planning, and collaboration!

The following visual depicts the 5 most popular design patterns employed in building AI agents.

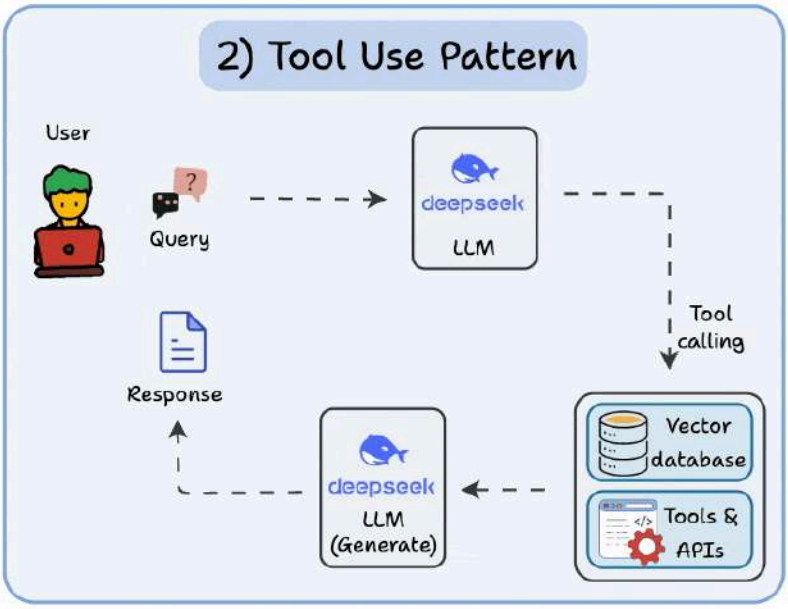


1) Reflection pattern



The AI reviews its own work to spot mistakes and iterate until it produces the final response.

2) Tool use pattern



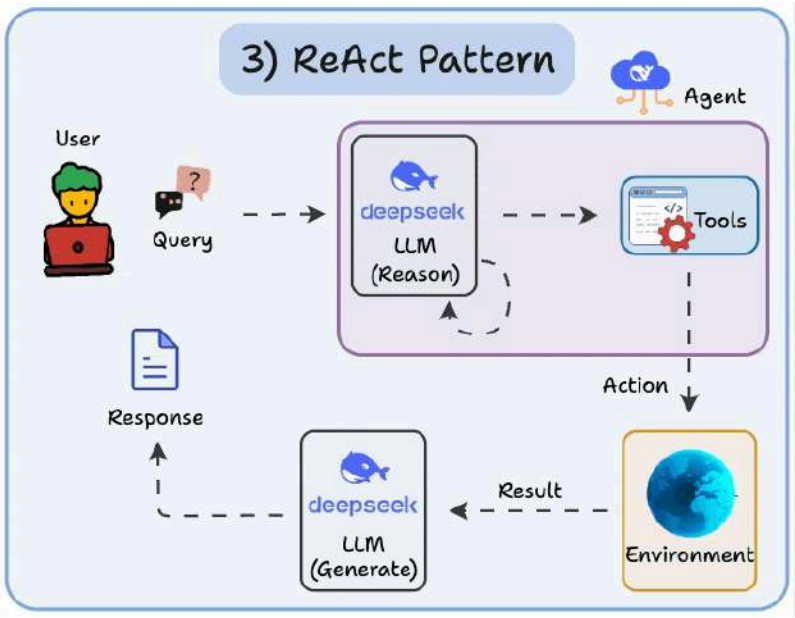
Tools allow LLMs to gather more information by:

- Querying a vector database
- Executing Python scripts

- Invoking APIs, etc.

This is helpful since the LLM is not solely reliant on its internal knowledge.

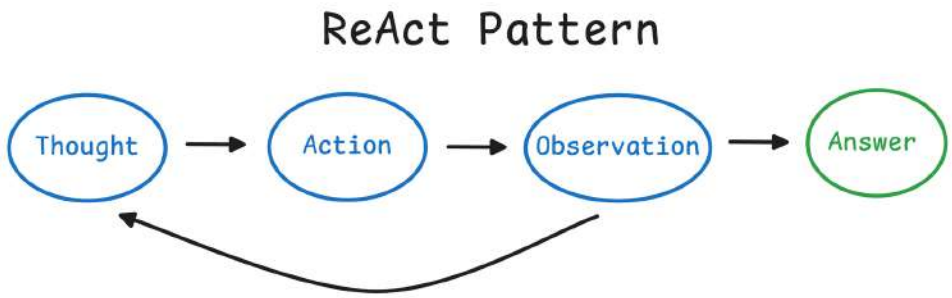
3) ReAct (Reason and Act) pattern



ReAct combines the above two patterns:

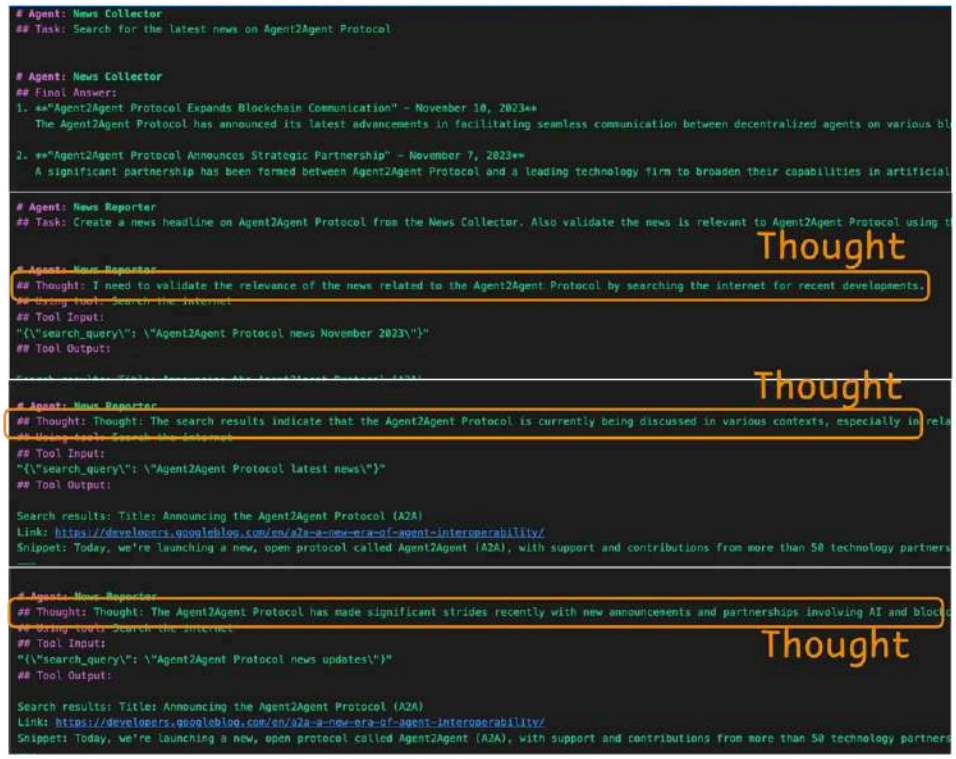
- The Agent reflects on the generated outputs.
- It interacts with the world using tools.

A ReAct agent operates in a loop of Thought → Action → Observation, repeating until it reaches a solution or a final answer. This is analogous to how humans solve problems:



Note: Frameworks like CrewAI primarily use this by default.

To understand this, consider the output of a multi-agent system below:



As shown above, the Agent is going through a series of thought activities before producing a response.

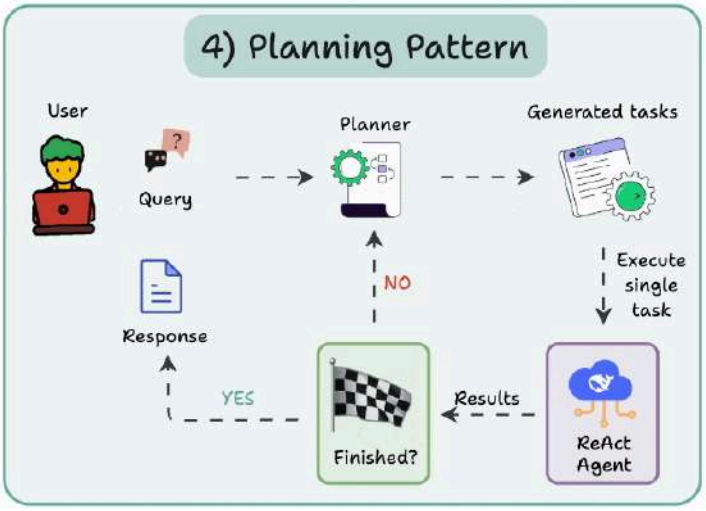
This is the ReAct pattern in action!

More specifically, under the hood, many such frameworks use the ReAct (Reasoning and Acting) pattern to let LLM think through problems and use tools to act on the world.

For example, an agent in CrewAI typically alternates between reasoning about a task and acting (using a tool) to gather information or execute steps, following the ReAct paradigm.

This enhances an LLM agent's ability to handle complex tasks and decisions by combining chain-of-thought reasoning with external tool use.

4) Planning pattern



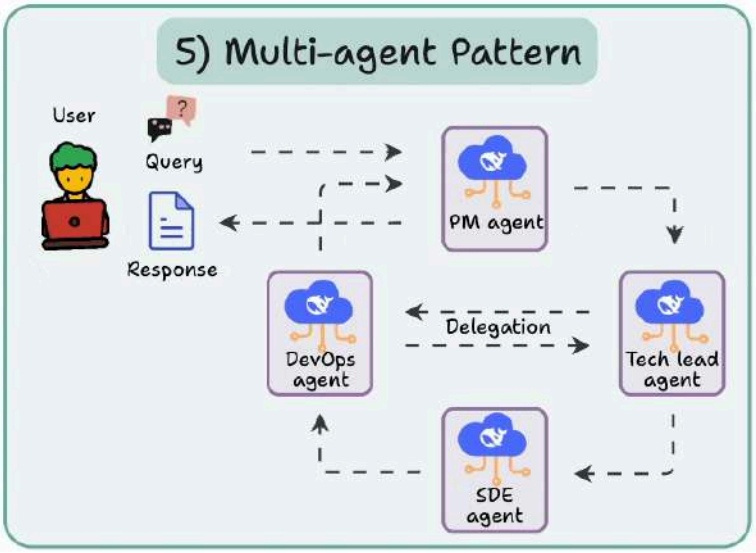
Instead of solving a task in one go, the AI creates a roadmap by:

- Subdividing tasks
- Outlining objectives

This strategic thinking solves tasks more effectively.

Note: In CrewAI, specify `planning=True` to use Planning.

5) Multi-Agent pattern



- There are several agents, each with a specific role and task.
- Each agent can also access tools.

All agents work together to deliver the final outcome, while delegating tasks to other agents if needed.

ReAct Implementation from Scratch

Below, we shall implement a ReAct Agent in two ways:

- Manually executing each step for better clarity.
- Without manual intervention to fully automate the Reasoning and Action process.

Let's look at the manual process first.

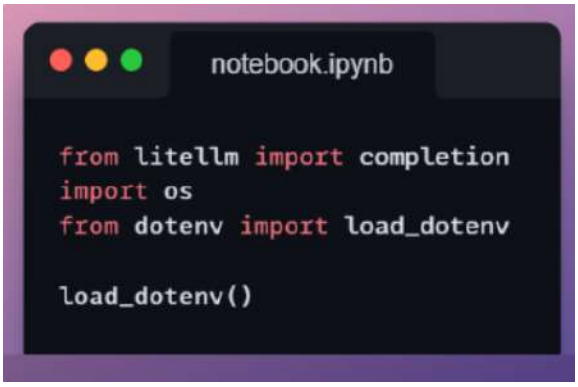
#1) ReAct with manual execution

In this section, we'll implement a lightweight ReAct-style agent from scratch, without using any orchestration framework like CrewAI or LangChain.

We'll manually simulate each round of the agent's reasoning, pausing, acting and observing exactly as a ReAct loop is meant to function.

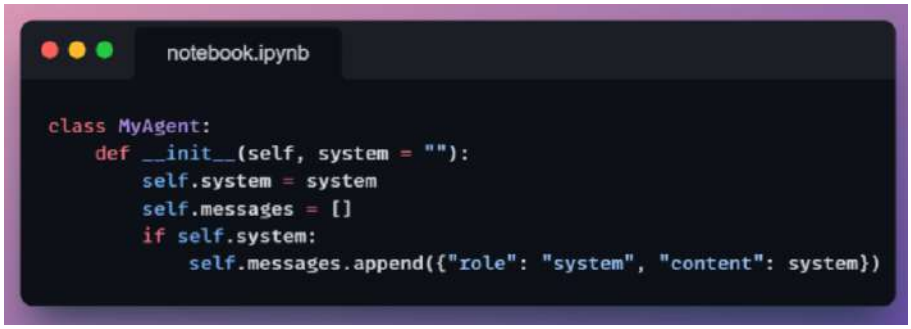
By running the logic cell-by-cell, we will gain full visibility and control over the thinking process, allowing us to debug and validate the agent's behavior at each step.

To begin, we load the environment variables (like your LLM API key) and import completion from LiteLLM (also install it first–pip install litellm), a lightweight wrapper to query LLMs like OpenAI or local models via Ollama.



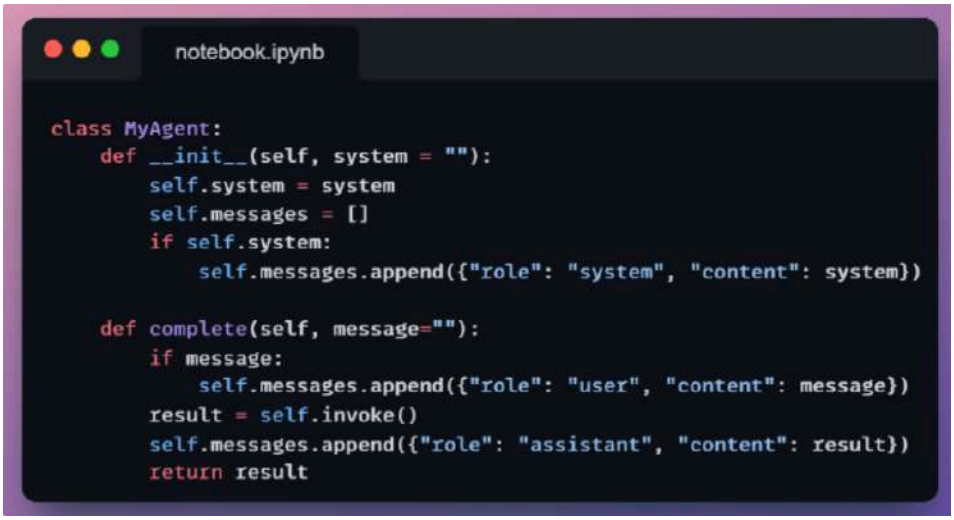
Next, we define a minimal Agent class, which wraps around a conversational LLM and keeps track of its full message history - allowing it to reason step-by-step, access system prompts, remember prior inputs and outputs, and produce multi-turn interactions.

Here's what it looks like:



- ***system*** (str): This is the system prompt that sets the personality and behavioral constraints for the agent. If passed, it becomes the very first message in the conversation just like in OpenAI Chat APIs.
- ***self.messages***: This list acts as the conversation memory. Every interaction, whether it's user input or assistant output is appended to this list. This history is crucial for LLMs to behave coherently across multiple turns.
- If ***system*** is provided, it's added to the message list using the special *"**role**": "**system**"* identifier. This ensures that every completion that follows is conditioned on the system instructions.

Next, we define a ***complete*** method in this class:



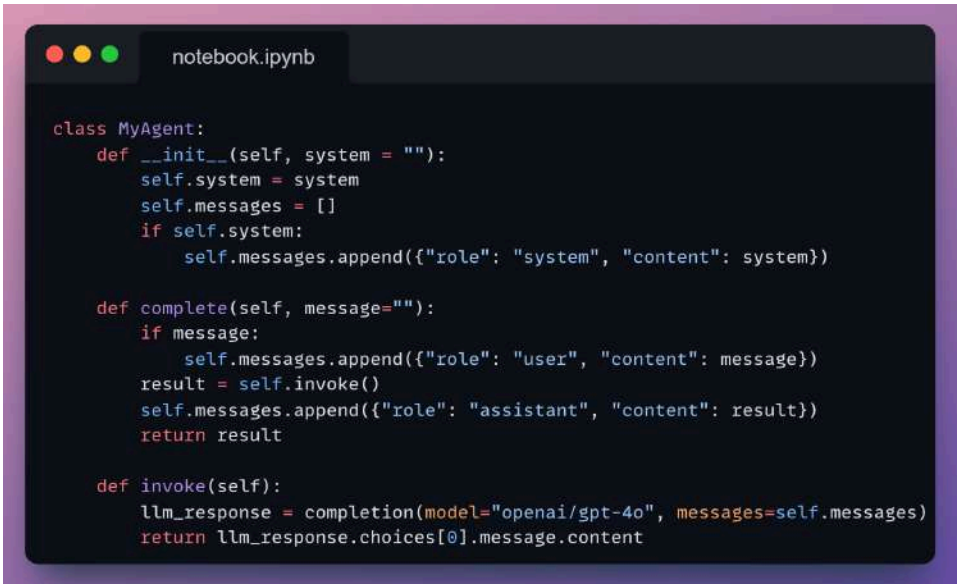
This is the core interface you'll use to interact with your agent.

- If a ***message*** is passed:
  - It gets appended as a *"**user**"* message to ***self.messages***.
  - This simulates the human asking a question or giving instructions.
- Then, ***self.invoke()*** is called (which we will define shortly). This method sends the full conversation history to the LLM.
- The model's reply (stored in ***result***) is then appended to ***self.messages*** as an *"**assistant**"* role.
- Finally, the reply is returned to the caller.

This method does three things in one call:

1. Records the user input.
2. Gets the model's reply.
3. Updates the message history for future turns.

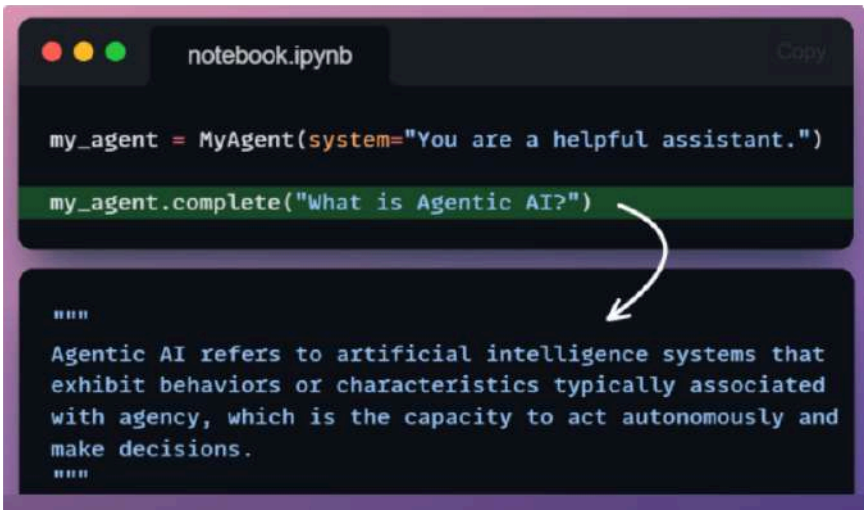
Finally, we have the ***invoke*** method below:



This method handles the actual API call to your LLM provider - in this case, via LiteLLM, using the *"**openai/gpt-4o**"* model.

- ***completion()*** is a wrapper around the chat completion API. It receives the entire message history and returns a response.
- We assume ***completion()*** returns a structure similar to OpenAI's format: a list of choices, where each choice has a ***.message.content*** field.
- We extract and return that content - the assistant's next response.

As a test, we can quickly run a simple interaction below:



At this stage, if we ask it about the previous message, we get the correct output, which shows the assistant has visibility on the previous context:

It correctly remembers and reflects!

Now that our conversational class is setup, we come to the most interesting part, which is defining a ReAct-style prompt.

Before an LLM can behave like an agent, it needs clear instructions - not just on what to answer, but how to go about answering. That's exactly what this ***system_prompt*** does, which is defined below:

notebook.ipynb

```
system_prompt = """
You run in a loop and do JUST ONE thing in a single iteration:

1) "Thought" to describe your thoughts about the input question.
2) "PAUSE" to pause and think about the action to take.
3) "Action" to decide what action to take from the list of actions available to you.
4) "PAUSE" to pause and wait for the result of the action.
5) "Observation" will be the output returned by the action.

At the end of the loop, you produce an Answer.

The actions available to you are:

math:
e.g. math: (14 * 5) / 4
Evaluates mathematical expressions using Python syntax.

lookup_population:
e.g. lookup_population: India
Returns the latest known population of the specified country.

Here's a sample run for your reference:

Question: What is double the population of Japan?

Iteration 1:
Thought: I need to find the population of Japan first.

Iteration 2:
PAUSE

Iteration 3:
Action: lookup_population: Japan

Iteration 4:
PAUSE

(you will now receive an output from the action)

Iteration 5:
Observation: 125,000,000

Iteration 6:
Thought: I now need to multiply it by 2.

Iteration 7:
Action: math: 125000000 * 2

Iteration 8:
PAUSE

(you will now receive an output from the action)

Iteration 9:
Observation: 250000000

Iteration 10:
Answer: Double the population of Japan is 250 million.

Whenever you have the answer, stop the loop and output it to the user.

Now begin solving:
""".strip()
```

This isn't just a prompt. It's a behavioral protocol - defining what structure the agent should follow, how it should reason, and when it should stop.

Let's break it down line by line.

You run in a loop and do JUST ONE thing in a single iteration:

This is the framing sentence. It tells the LLM not to rush toward an answer. Instead, it should proceed step by step, following a defined pattern in a loop - mirroring how a ReAct agent works.

1) "Thought" to describe your thoughts about the input question.

2) "PAUSE" to pause and think about the action to take.

3) "Action" to decide what action to take from the list of actions available to you.

4) "PAUSE" to pause and wait for the result of the action.

5) "Observation" will be the output returned by the action.

Here, we give the LLM a reasoning template. These are the same primitives found in all ReAct-style agents.

Let's break each down:

- Thought: The agent's internal monologue. What is it currently thinking about?
- PAUSE (1): Instead of jumping to action, this forces the model to take a breath - simulating asynchronous steps in a multi-agent environment.
- Action: The agent picks from the list of tools it is given.
- PAUSE (2): Wait again, this time for the actual tool result.
- Observation: This will be injected into the prompt by you (the controller or human), after the tool runs.

By splitting this into explicit parts, we avoid hallucinations and ensure the agent works in a controlled loop.

At the end of the loop, you produce an Answer.

This tells the agent: once it has all the required information - break the loop and give the final answer. No need to keep reasoning indefinitely.

The actions available to you are:

math:

*e.g. math: (14 * 5) / 4*

Evaluates mathematical expressions using Python syntax.

lookup_population:

e.g. lookup_population: India

Returns the latest known population of the specified country.

This is a mini API reference for the agent. We show:

- The name of each tool.
- How to invoke it.
- What kind of output it produces.

This is critical. Without a clear spec, the LLM might:

- Invent non-existent tools.
- Use incorrect syntax.
- Misinterpret what the tool is supposed to do.

By using clear formatting and examples, we teach the model how to interface with tools in a safe, predictable way.

Here's a sample run for your reference:

Question: What is double the population of Japan?

Iteration 1:

Thought: I need to find the population of Japan first.

Iteration 2:

PAUSE

...

Iteration 9:

Observation: 250000000

Iteration 10:

Answer: Double the population of Japan is 250 million.

This worked-out example gives the LLM a pattern to follow. Even more importantly, it provides the developer (you) a way to intervene at each step - injecting tool results or validating whether the flow is working correctly.

With this sample trace:

- The agent knows how to think.
- The agent knows how to act.
- The agent knows when to stop.

Whenever you have the answer, stop the loop and output it to the user.

Now begin solving:

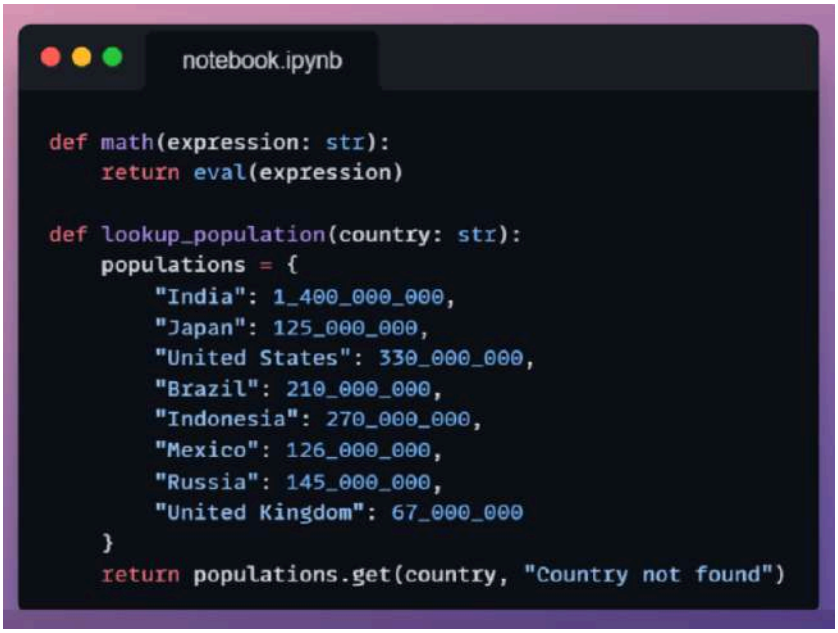
These closing lines are essential.

Without this explicit stop signal, the LLM might continue indefinitely. You're telling it: "When you have all the puzzle pieces, just say the answer and exit the loop."

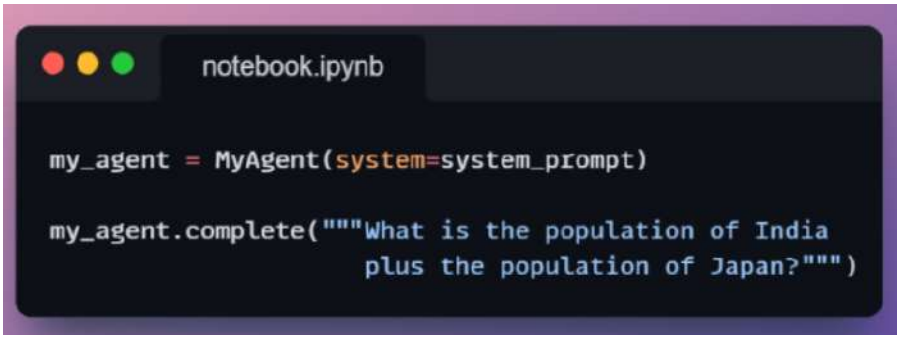
The power of this ***system_prompt*** lies in its structure:

- It models intelligent behavior, not just question answering.
- It imposes strong constraints: think before acting, act within defined bounds, and wait for observations.
- It separates reasoning from execution, mimicking how humans operate.
- It creates a feedback-friendly iteration loop for multi-step problems.

Now that the prompt is defined, we implement the tools.



Finally, we begin a manual ReAct session:

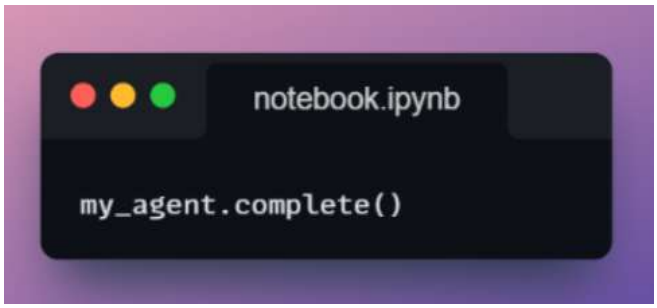


This produces the following output:

Iteration 1:

Thought: I need to find the population of India first.

We, as a user, don't have any input to give at this stage so we just invoke the ***complete()*** method again:

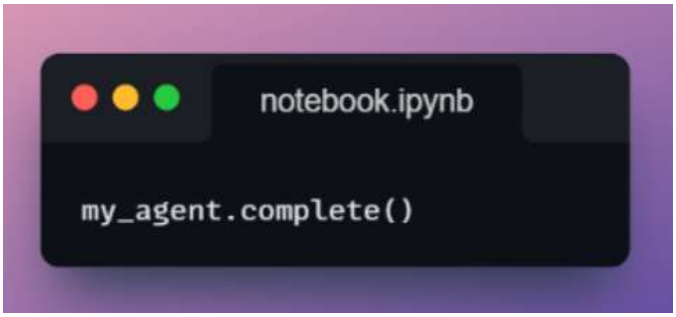


This produces the following output:

Iteration 2:

PAUSE

Yet again, we, as a user, don't have any input to give at this stage so we just invoke the ***complete()*** method again:



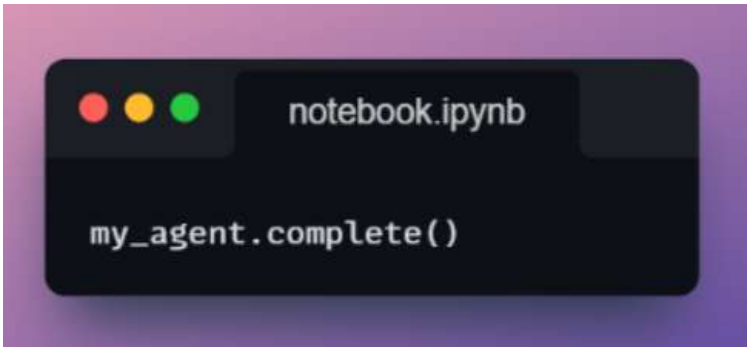
This produces the following output:

Iteration 3:

Action: lookup_population: India

Now it wants to act.

We still don't have any input to give at this stage so we just invoke the ***complete()*** method again:

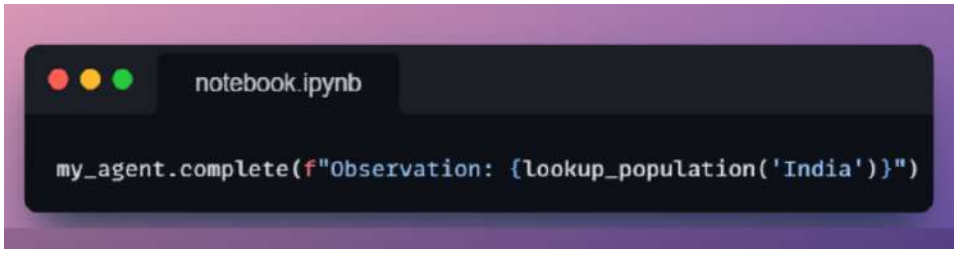


This produces the following output:

Iteration 4:

PAUSE

At this stage, it needs to get the tool output in the form of an observation. Here, let's intervene and provide it with the observation:

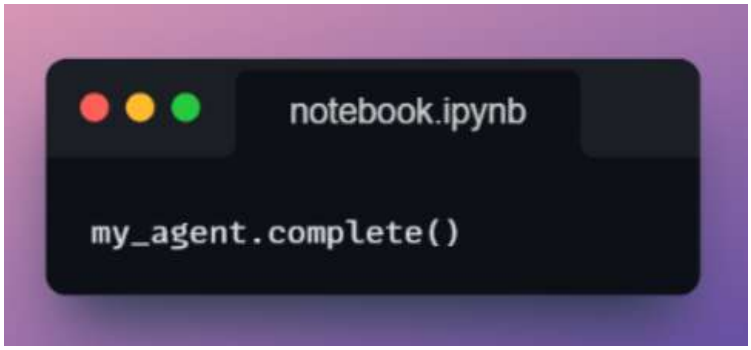


This produces the following output:

Iteration 5:

Thought: Now I need to find the population of Japan.

We let it continue its execution:

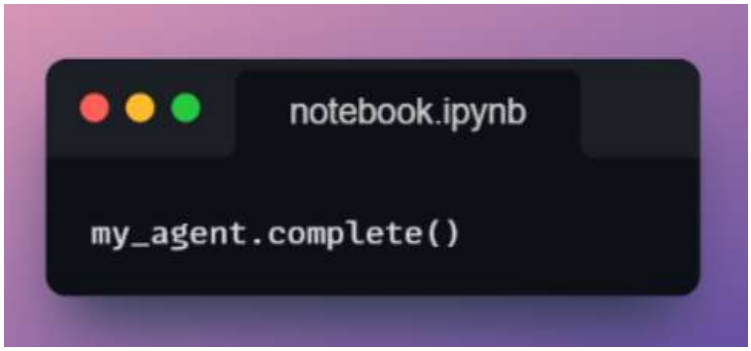


This produces the following output:

Iteration 6:

PAUSE

We again let it continue its execution:



We get the following output:

Iteration 7:

Action: lookup_population: Japan

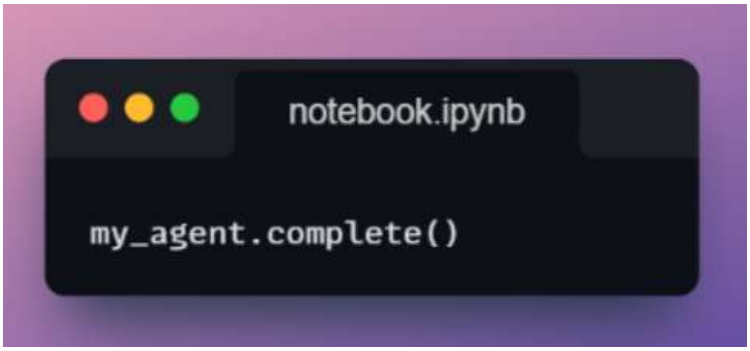
At this stage, it needs to get the tool output in the form of an observation. Here, let's again intervene and provide it with the observation:

This produces the following output:

Iteration 8:

Thought: I now have the populations of both India and Japan. I need to add them together.

We again let it continue its execution:

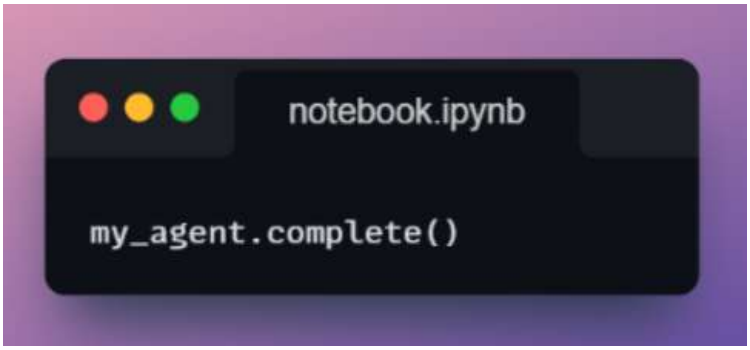


We get the following output:

Iteration 9:

Action: math: 1400000000 + 125000000

Now we should expect a pause according to the pattern specified:



Iteration 10:

PAUSE

It is again seeking an observation, which is the sum of Japan's population and India's population. To do this, we again manually intervene and provide it with the output:

Finally, in this iteration, we get the following output:

Iteration 11:

Answer: The sum of the population of India and the population of Japan is 1,525,000,000.

Great!!

With this process:

- The LLM thought about what steps to take.
- It chose actions to execute.
- We manually injected tool outputs like real-world observations.

- It looped until it had enough information to generate a final answer.

This gives us an explicit understanding of how reasoning and actions come together in ReAct-style agents.

In the next part, we'll fully automate this - no manual calls required and build a full controller that simulates this entire loop programmatically.

#2) ReAct without manual execution

Now that we have understood how the above ReAct execution went, we can easily automate that to remove our interventions.

In this section, we'll create a controller function that:

- Sends an initial question to the agent,
- Reads its thoughts and actions step-by-step,
- Automatically runs external tools when asked,
- Feeds back observations to the agent,
- And stops the loop once a final answer is found.

This is the entire code that does this:

Let's break down the full loop.

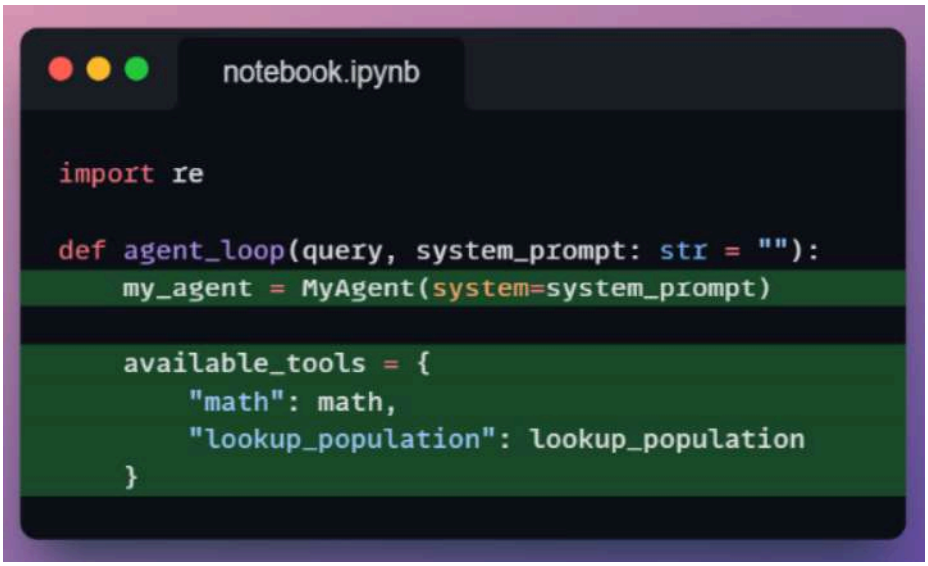
We begin by defining the ***agent_loop()*** function:

It takes:

- ***query***: the user's natural language question.

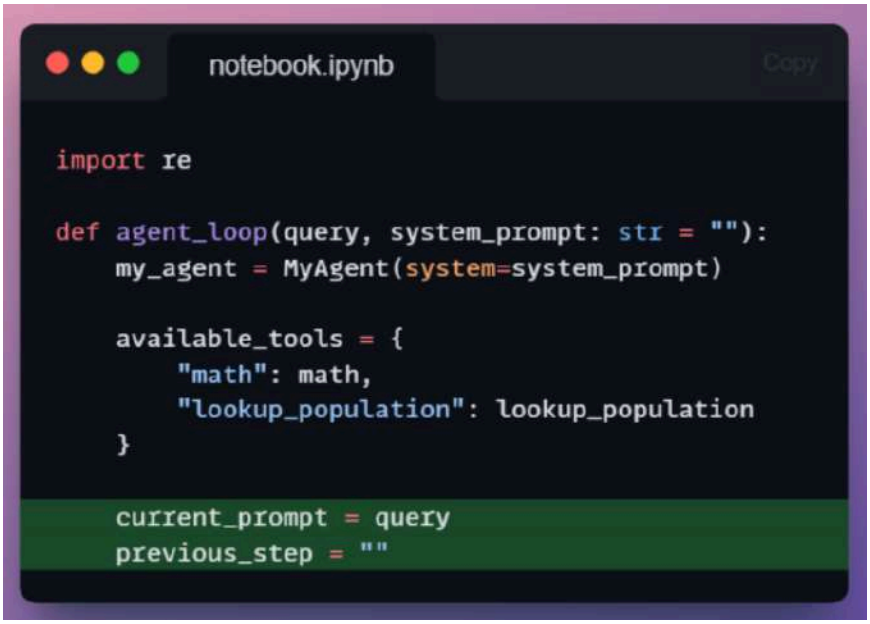
- ***system_prompt***: the same ReAct system prompt we explored earlier (defining the behavior loop).

Next, inside this function, we initialize the Agent and available tools:



- Create a new MyAgent instance, using the structured ReAct prompt.
- Define the dictionary of callable tools available to the agent. These names must match exactly what the agent uses in its ***Action:*** lines.

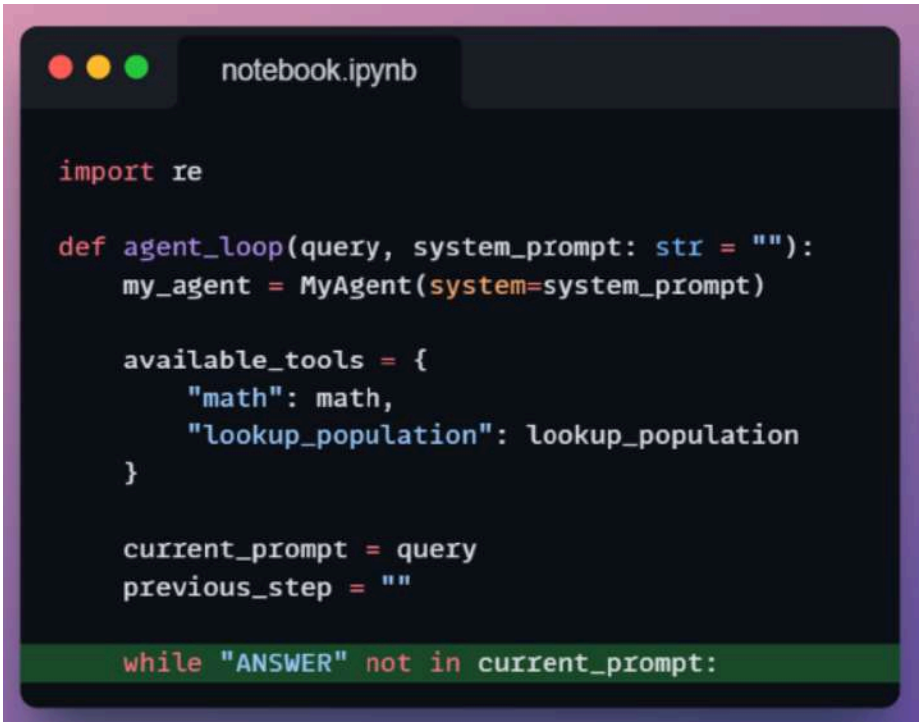
Moving on, we defined some state variables:



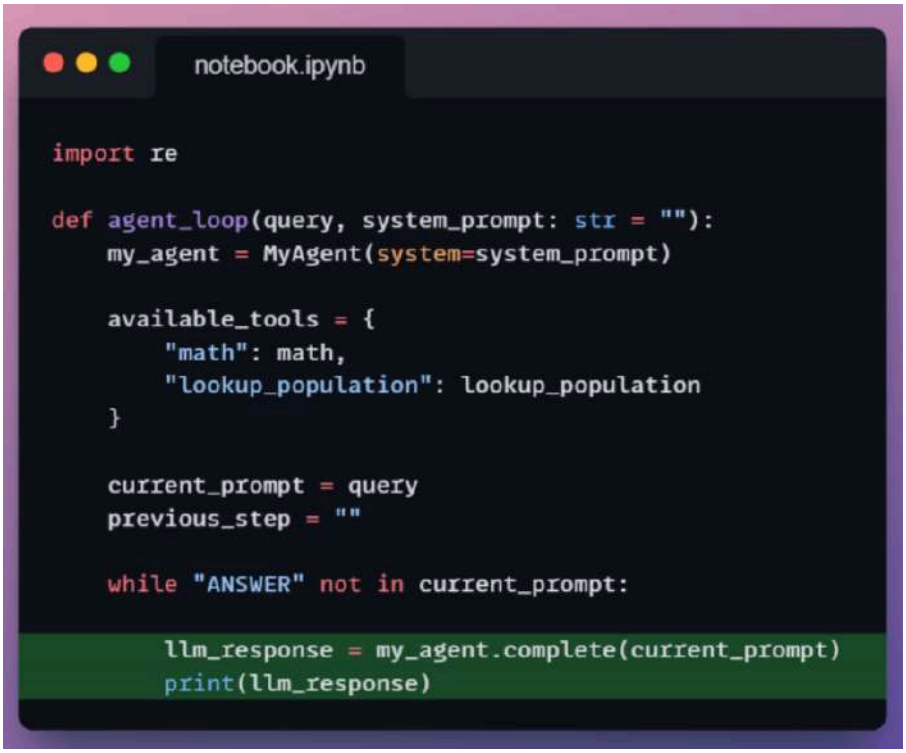
current_prompt stores the next message to be sent to the LLM.

previous_step helps track the last stage (e.g., Thought, Action) for better control flow.

Next, we run the reasoning loop, which continues until the agent produces a final answer. The answer is expected to be marked with ***Answer:*** based on our prompt design:



Next, we feed the ***current_prompt*** into the agent.

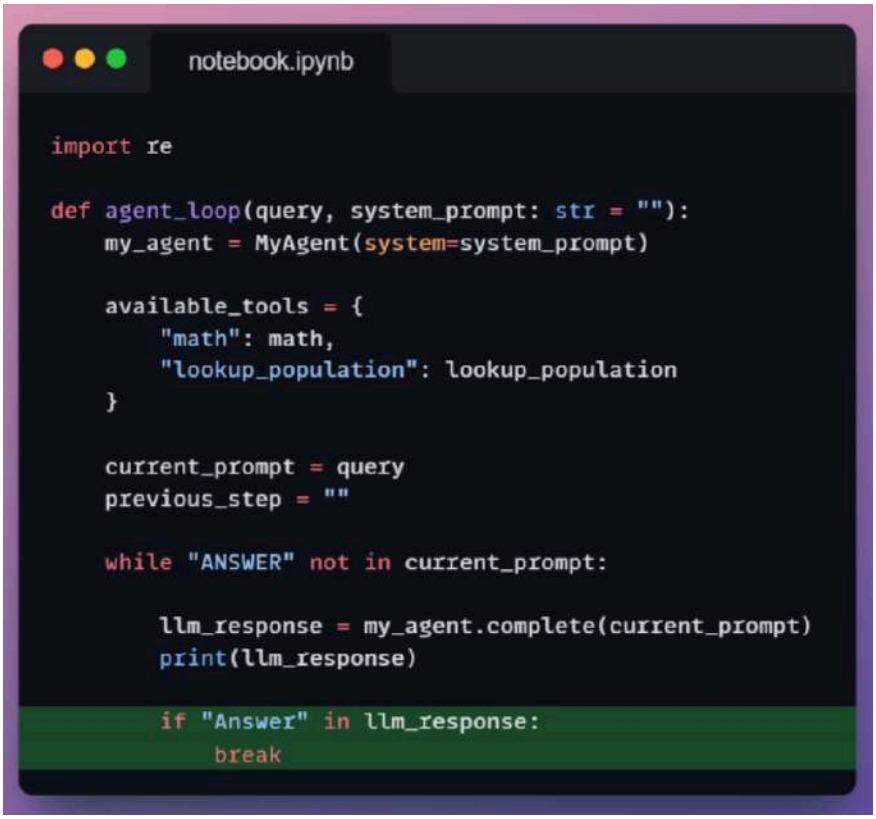


The ***current_prompt*** could be:

- The initial user query,
- A blank string to let the agent continue reasoning,
- An observation from a tool.

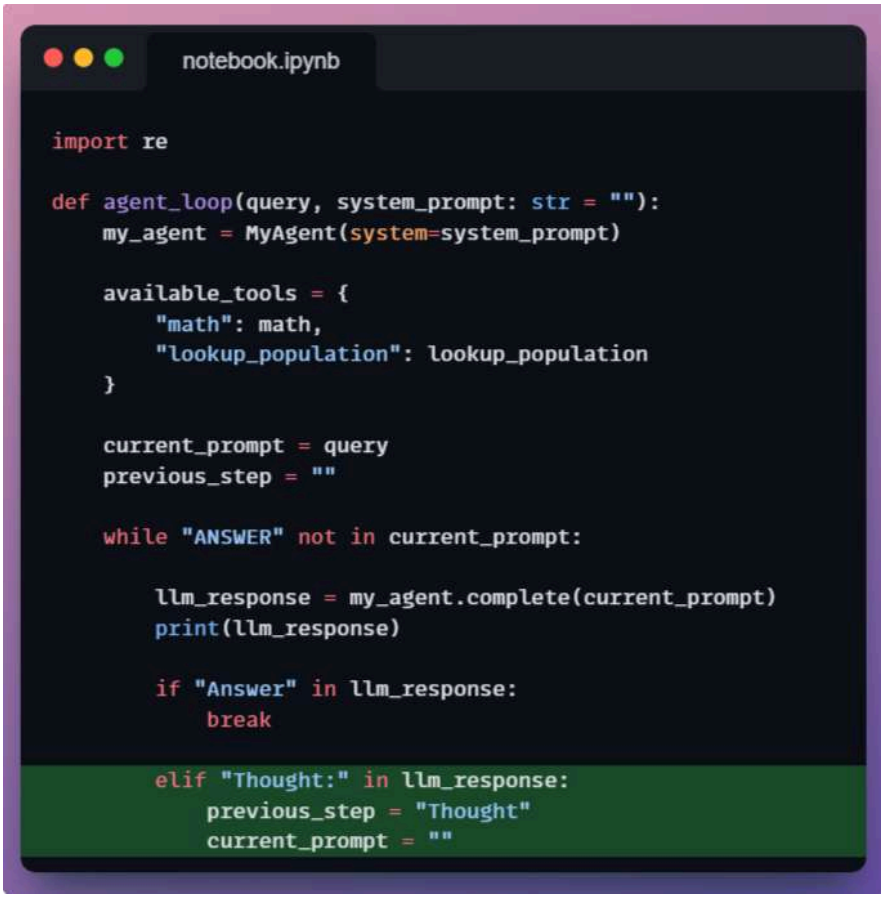
We then print the agent's output, so we can inspect each iteration.

Next, if the agent produces a final answer, we break the loop.

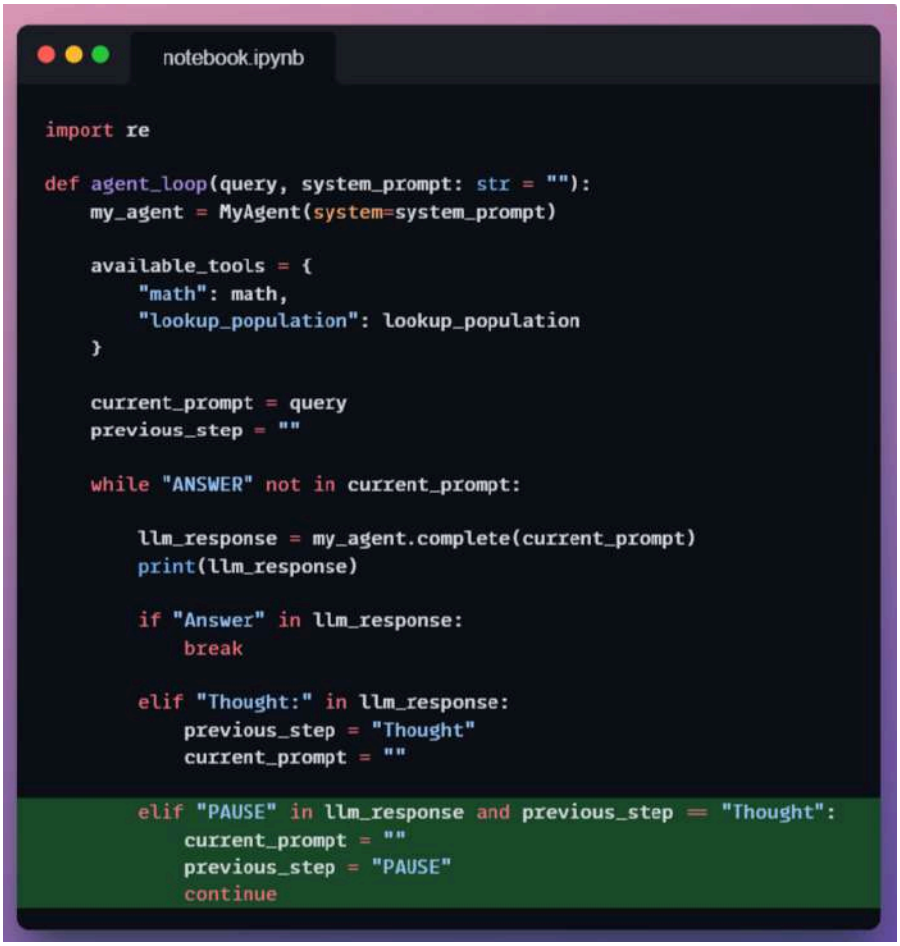


In another case, if the response includes a ***Thought:*** line, we:

- Record the step type as "Thought".
- Set ***current_prompt*** to an empty string to continue to the next stage (a PAUSE).

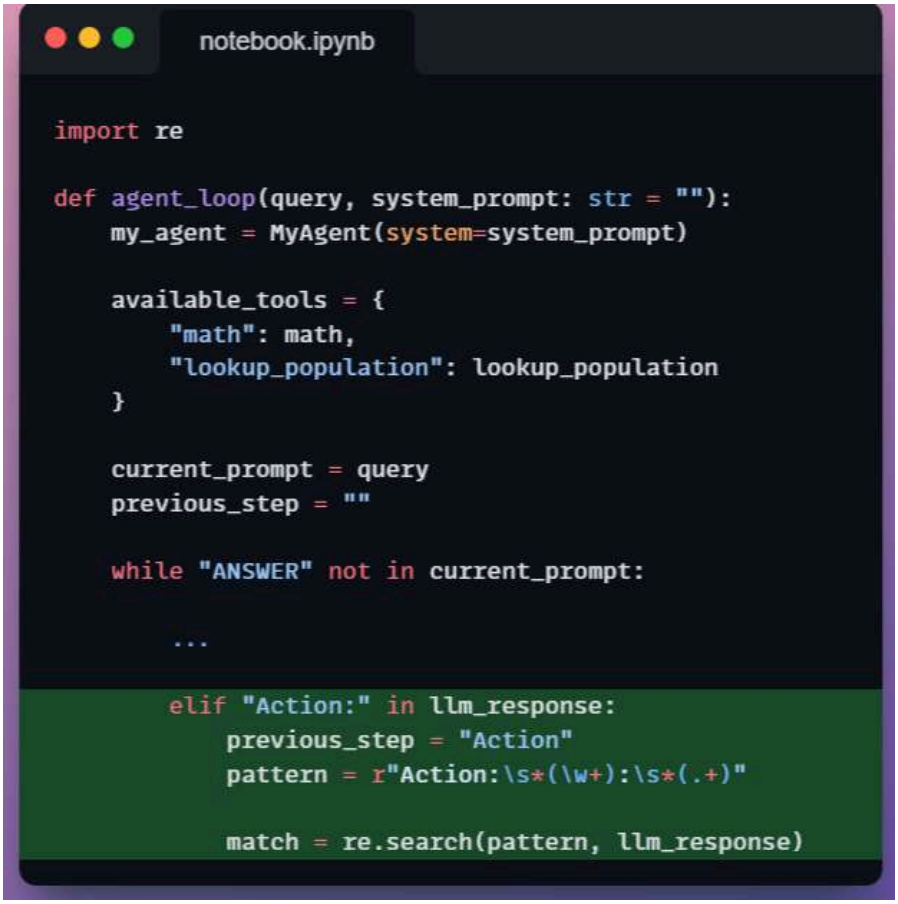
- 

Next, we catch the first PAUSE right after the Thought. Nothing else needs to be done here - we just move to the next step.



If we detect an ***Action:*** line, we:

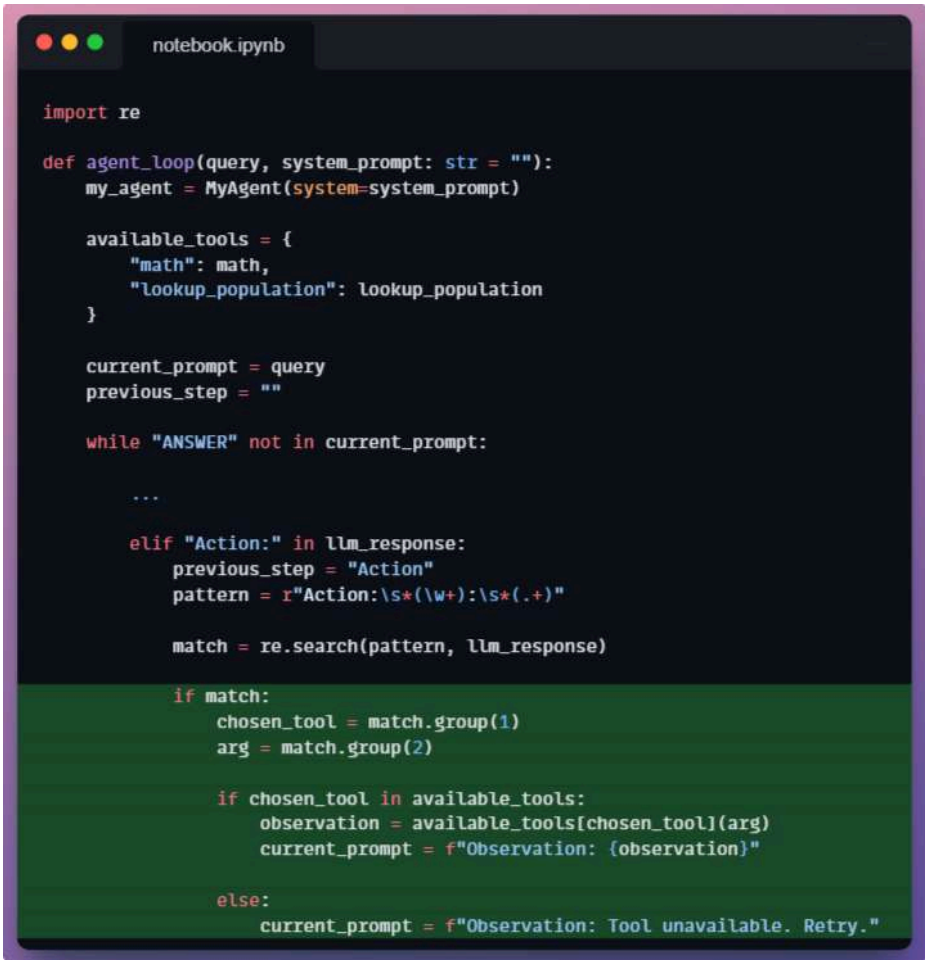
- Note that we're in the action step.
- Use a regex to extract the tool name and its argument.

- 

For example, in: ***Action: lookup_population: India***, the regex pulls out:

- ***lookup_population*** as the tool.
- ***India*** as the argument.

Moving on, we execute the tool and capture the observation:

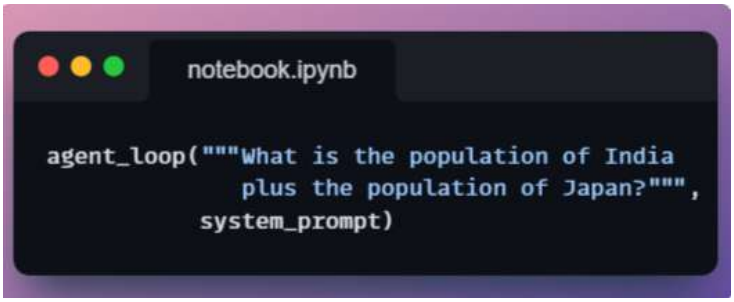


- If the tool name is valid, we call it like a Python function and capture the result.
- We format the output into Observation: ... so the agent can use it in the next step.
- If the tool doesn't exist, we ask the agent to retry.

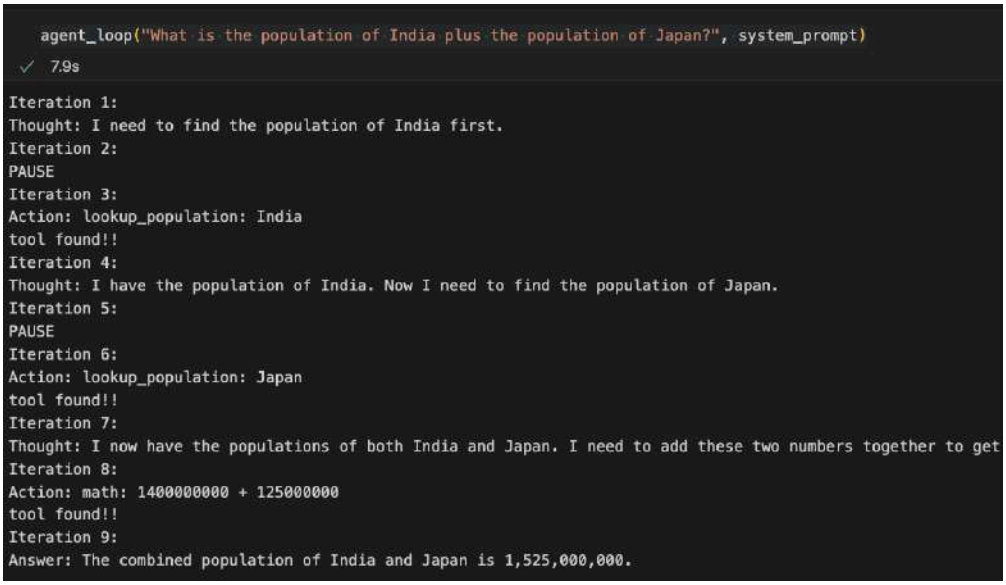
This mimics tool execution + response injection.

Done!

Now we can run this function as follows:



This produces the following output, which is indeed correct:



You now have a fully working ReAct loop without needing any external framework.

Of course, In this implementation, we're using regex matching and hardcoded conditionals to parse the agent's actions and route them to the correct tools.

This approach works well for a tightly controlled setup like this demo. However, it's brittle:

- If the agent slightly deviates from the expected format (e.g., adds extra whitespace, uses different casing, or mislabels an action), the regex could fail to match.

- We're also assuming that the agent will never call a tool that doesn't exist, and that all tools will succeed silently.

In a production-grade system, you'd want to:

- Add more robust parsing (e.g., structured prompts with JSON outputs or function calling).
- Include tool validation, retries, and exception handling.
- Use guardrails or output formatters to constrain what the LLM is allowed to emit.

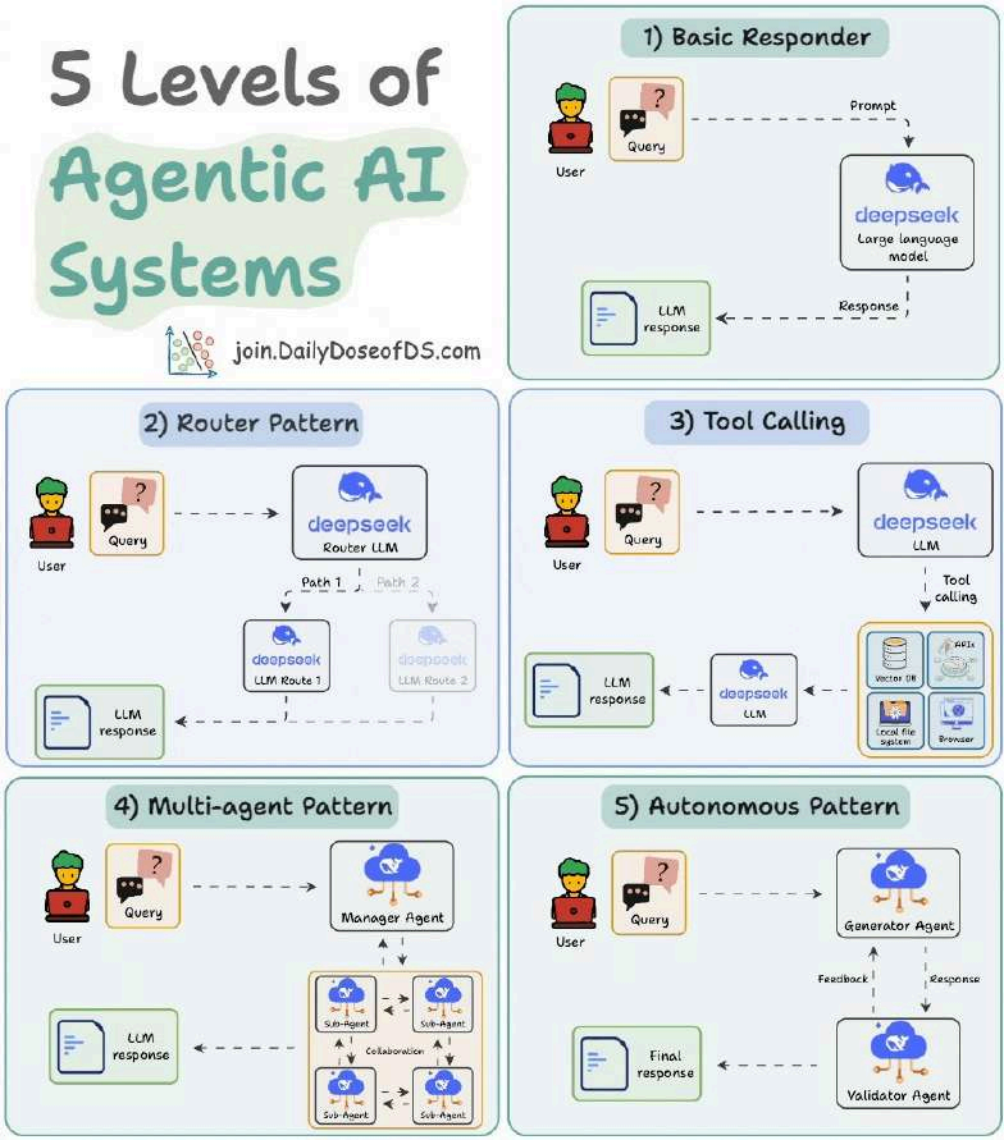
But for the purpose of understanding how ReAct-style loops work under the hood, this is a clean and minimal place to start. It gives you complete transparency into what's happening at each stage of the agent's reasoning and execution process.

This loop demonstrates how a simple agent can think, act, and observe, all powered by your own Python + local LLM stack.

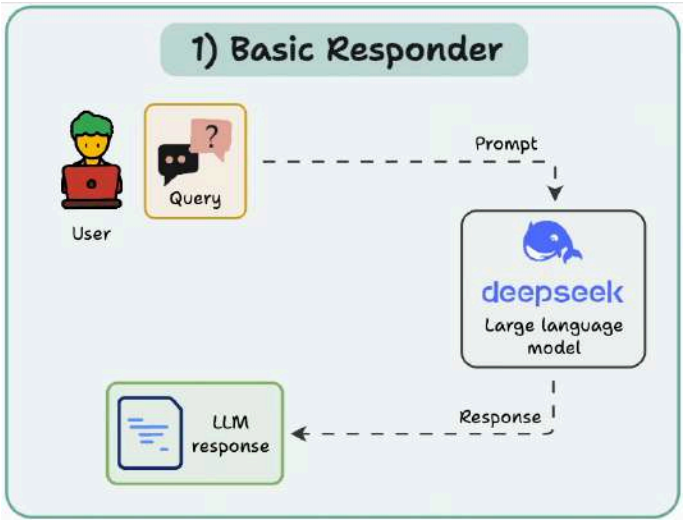
5 Levels of Agentic AI Systems

Agentic AI systems don't just generate text; they can make decisions, call functions, and even run autonomous workflows.

The visual explains 5 levels of AI agency - from simple responders to fully autonomous agents.



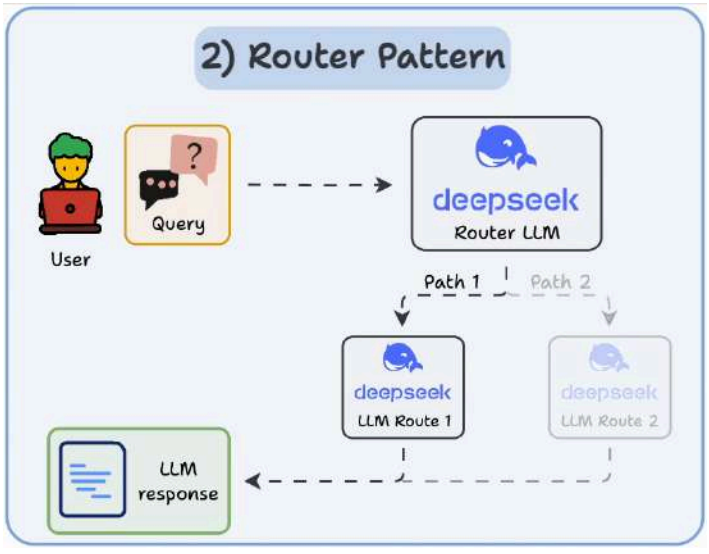
1) Basic responder



A human guides the entire flow.

The LLM is just a generic responder that receives an input and produces an output. It has little control over the program flow.

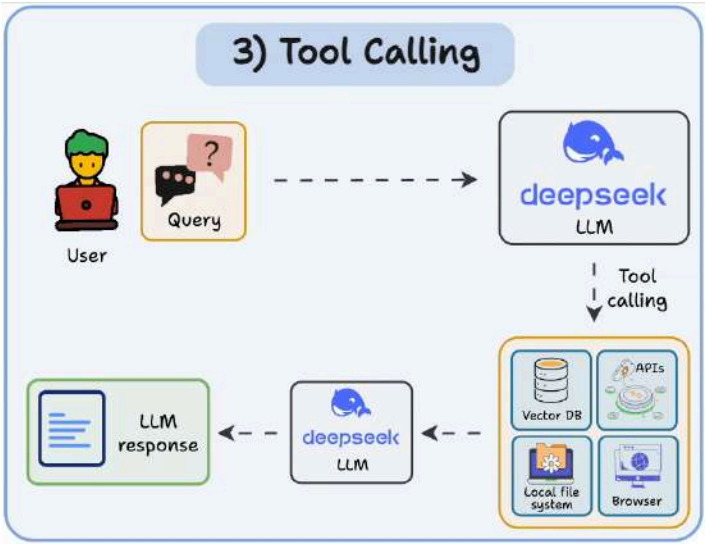
2) Router pattern



A human defines the paths/functions that exist in the flow.

The LLM makes basic decisions on which function or path it can take.

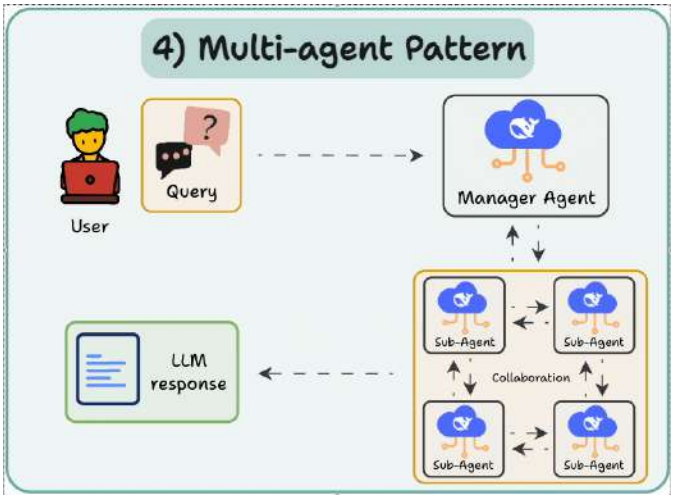
3) Tool calling



A human defines a set of tools the LLM can access to complete a task.

LLM decides when to use them and also the arguments for execution.

4) Multi-agent pattern

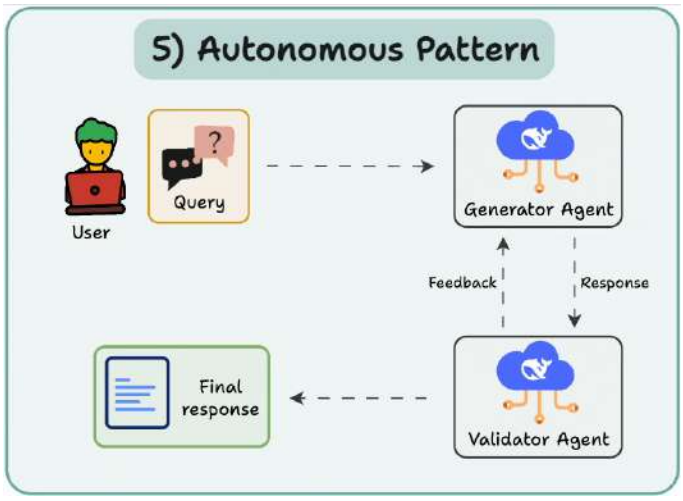


A manager agent coordinates multiple sub-agents and decides the next steps iteratively.

A human lays out the hierarchy between agents, their roles, tools, etc.

The LLM controls execution flow, deciding what to do next.

5) Autonomous pattern

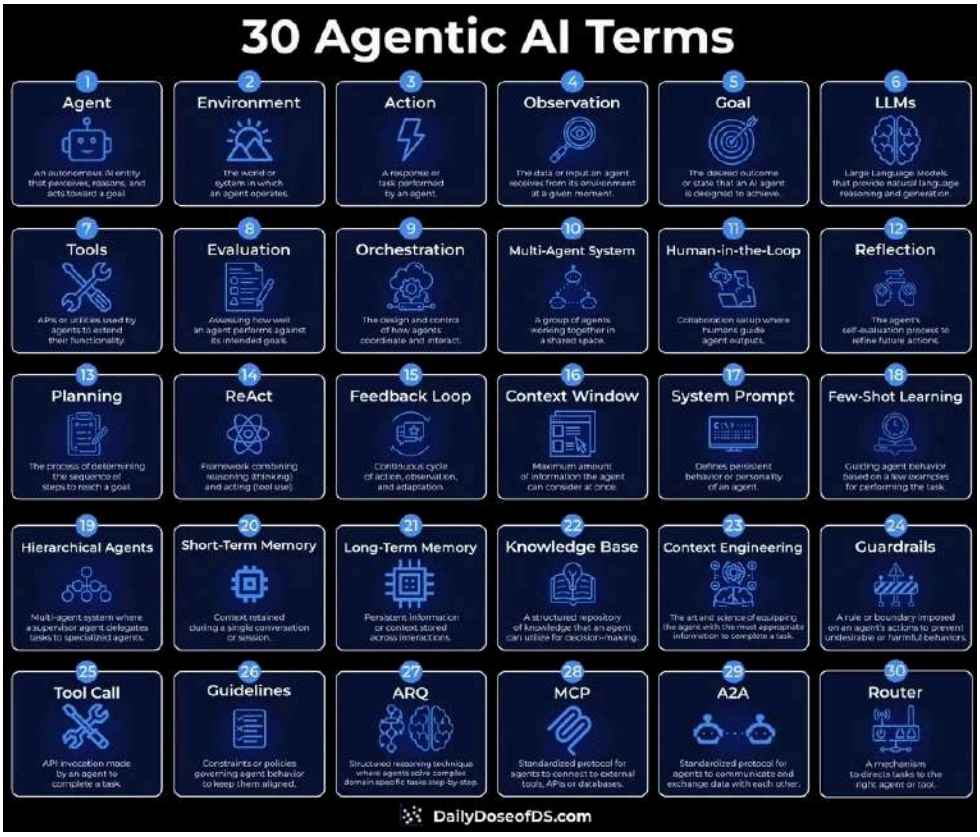


The most advanced pattern, wherein, the LLM generates and executes new code independently, effectively acting as an independent AI developer.

30 Must-Know Agentic AI Terms

We put together a quick visual guide to the 30 most important terms in Agentic AI, covering some of the most essential things you need to understand about how modern AI agents actually think, act, and collaborate.

If you've been exploring agent frameworks like CrewAI, LangGraph, or AutoGen, this glossary will help you connect the dots between key building blocks.



Agent: An autonomous AI entity that perceives, reasons, and acts toward a goal (covered with full implementations here).

Environment: The world or system in which an agent operates and interacts.

Action: A response or task performed by an agent based on its reasoning or goals.

Observation: The data or input an agent receives from its environment at any given moment.

Goal: The desired outcome that an Agent is designed to achieve.

LLMs: Large Language Models that enable agents to reason and generate natural language.

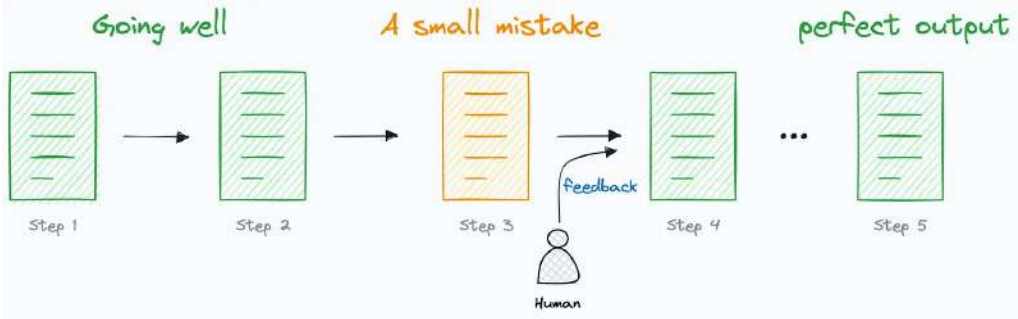
Tools: APIs or utilities agents use to extend their functionality and capabilities to interact with the world.

Evaluation: The process of assessing how well an agent performs against its intended goals (covered here with implementation).

Orchestration: The coordination and control of multiple agents working together to achieve complex tasks.

Multi-agent system: A group of agents collaborating to accomplish a final goal (implemented from scratch in pure Python here).

Human-in-the-loop: A setup where humans intervene or guide the agent's decision-making process.



Reflection: The agent's process of self-assessing its actions to improve future performance.

Planning: Determining the sequence of steps an agent must take to reach its goal (implemented from scratch in pure Python here).

ReAct: A framework where reasoning (thought) and acting (tool use) are combined step by step (implemented from scratch in pure Python here).

Feedback loop: A continuous process of collecting outcomes, observing effects, and adjusting actions.

Context window: The maximum amount of information an agent can consider at once.

System prompt: The persistent background instructions or personality that define an agent's behavior.

Few-shot learning: Teaching an agent new behaviors or tasks with just a few examples.

Hierarchical Agents: A multi-level agent structure where a supervisor agent delegates tasks to sub-agents.

Short-term memory: Temporary context stored during a single session or conversation.

Long-term memory: Persistent context stored across multiple sessions for continuity and learning (covered in detail here with code).

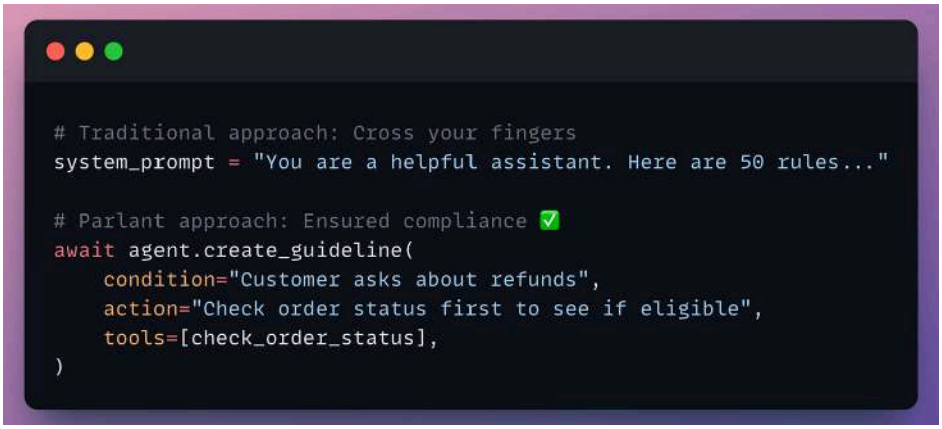
Knowledge base: A structured repository of information that agents can use for reasoning and decision-making (covered in detail here with code).

Context engineering: The practice of shaping what information an agent sees to optimize its output (here's a demo we covered).

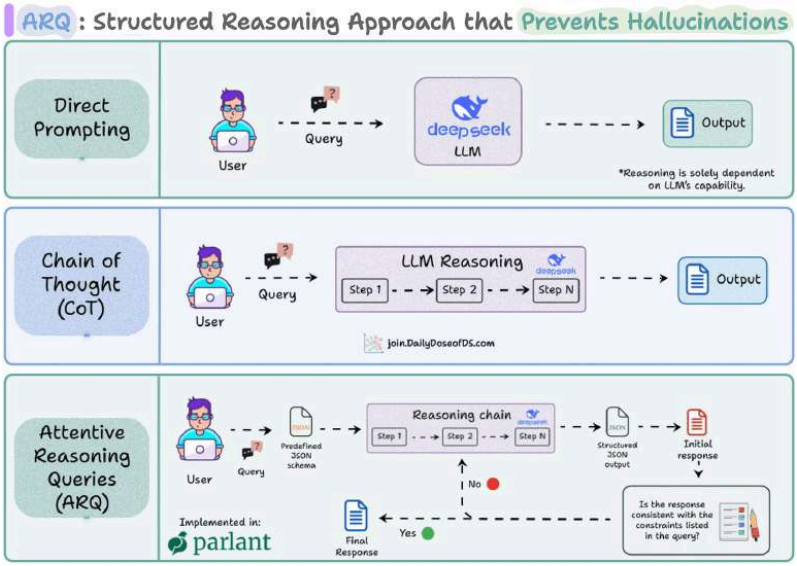
Guardrails: Rules or boundaries that prevent an agent from taking harmful or undesired actions (covered with code here).

Tool call: An API invocation made by an agent to perform a specific task.

Guidelines: Policies or constraints that keep an agent's behavior aligned with desired outcomes.

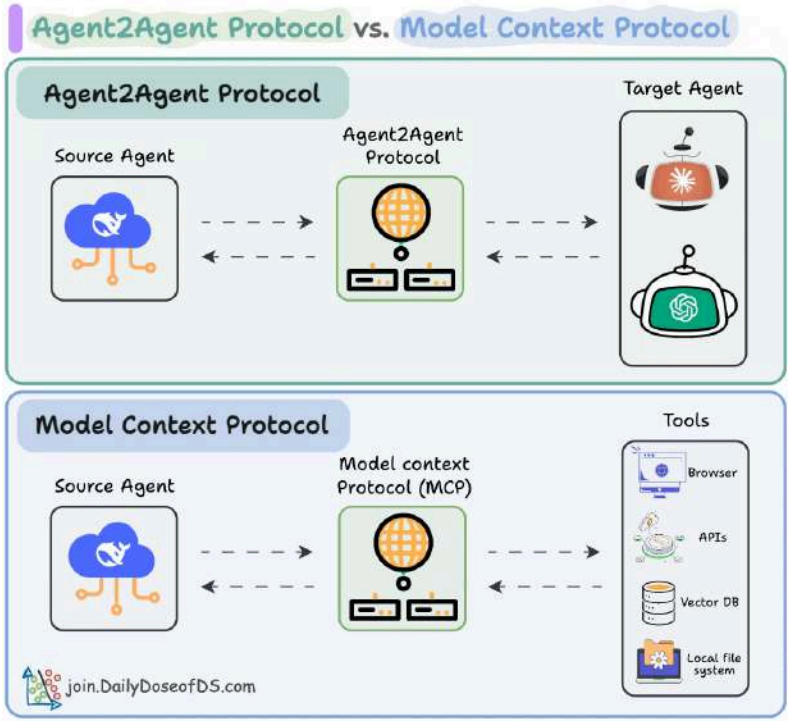


ARQ: A new structured reasoning approach where an agent solves complex, domain-specific problems step by step (covered here).



MCP: A standardized way for agents to connect to external tools, APIs, and data sources (learn how to build MCP servers, MCP clients, JSON-RPC, Sampling, Security, Sandboxing in MCPs, and using LangGraph/LlamaIndex/CrewAI/PydanticAI with MCP here).

A2A: Agent-to-Agent protocol enabling agents to communicate and exchange data directly (here's a visual guide).

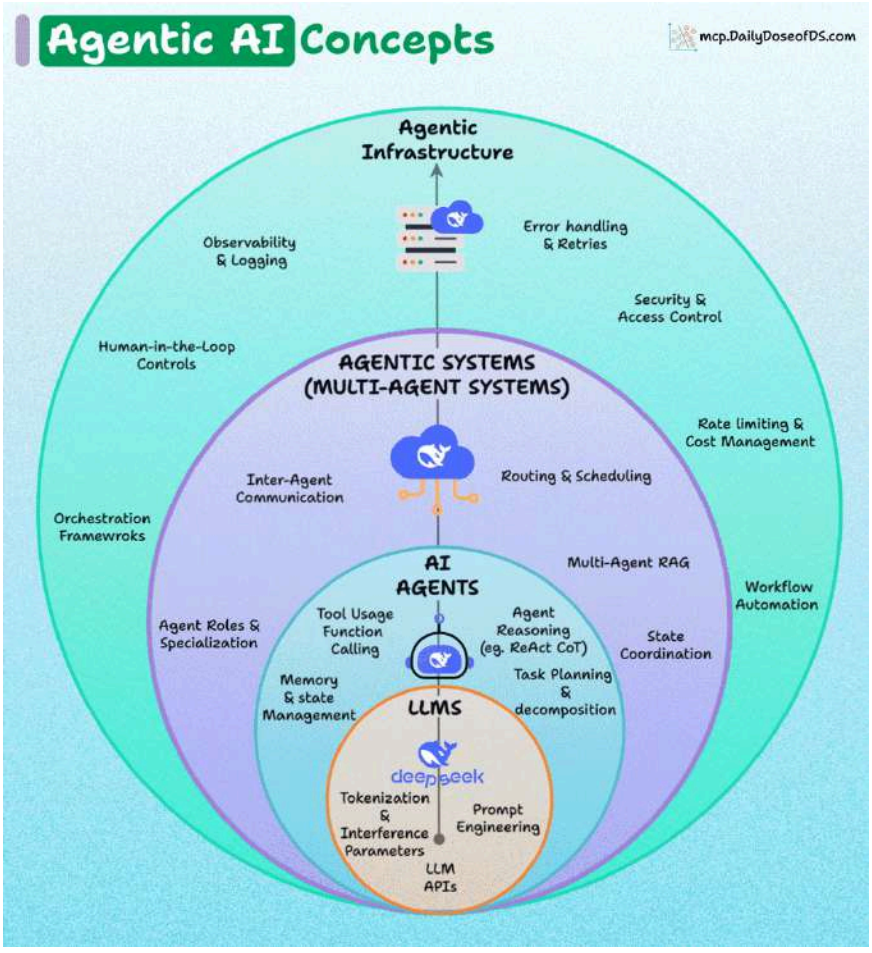


Router: A mechanism that directs tasks to the most appropriate agent or tool.

Each of these terms forms a key piece of the agentic AI ecosystem that AI engineers should know.

4 Layers of Agentic AI

The following graphic depicts a layered overview of Agentic AI concepts, depicting how the ecosystem is structured from the ground up (LLMs) to higher-level orchestration (Agentic Infrastructure).



Let's break it down layer by layer:

1) LLMs (foundation layer)

At the core, you have LLMs like GPT, DeepSeek, etc.

Core concepts here:

- Tokenization & inference parameters: how text is broken into tokens and processed by the model.
- Prompt engineering: designing inputs to get better outputs.
- LLM APIs: programmatic interfaces to interact with the model.

This is the engine that powers everything else.

2) AI Agents (built on LLMs)

Agents wrap around LLMs to give them the ability to act autonomously.

Key responsibilities:

- Tool usage & function calling: connecting the LLM to external APIs/tools.
- Agent reasoning: reasoning methods like ReAct (reasoning + act) or Chain-of-Thought.
- Task planning & decomposition: breaking a big task into smaller ones.
- Memory management: keeping track of history, context, and long-term info.

Agents are the brains that make LLMs useful in real-world workflows.

3) Agentic systems (multi-agent systems)

When you combine multiple agents, you get agentic systems.

Features:

- Inter-Agent communication: agents talking to each other, making use of protocols like ACP, A2A if needed.
- Routing & scheduling: deciding which agent handles what, and when.
- State coordination: ensuring consistency when multiple agents collaborate.
- Multi-Agent RAG: using retrieval-augmented generation across agents.

- Agent roles & specialization: Agents with unique purposes
- Orchestration frameworks: tools (like CrewAI, etc.) to build workflows.

This layer is about collaboration and coordination among agents.

4) Agentic Infrastructure

The top layer ensures these systems are robust, scalable, and safe.

This includes:

- Observability & logging: tracking performance and outputs (using frameworks like DeepEval).
- Error handling & retries: resilience against failures.
- Security & access control: ensuring agents don't overstep.
- Rate limiting & cost management: controlling resource usage.
- Workflow automation: integrating agents into broader pipelines.
- Human-in-the-loop controls: allowing human oversight and intervention.

This layer ensures trust, safety, and scalability for enterprise/production environments.

Overall, Agentic AI, as a whole, involves a stacked architecture, where each outer layer adds reliability, coordination, and governance over the inner layers.

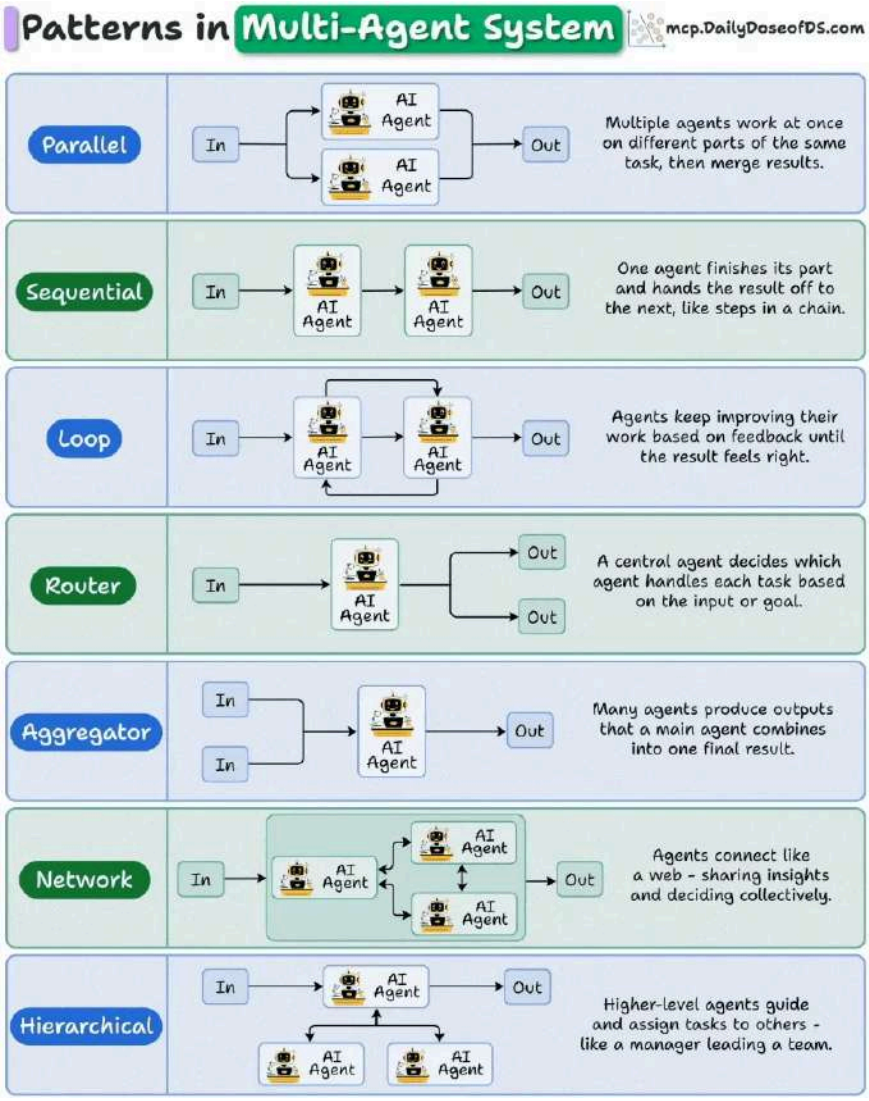
7 Patterns in Multi-Agent Systems

Monolithic agents (single LLMs stuffed with system prompts) didn't sustain for too long.

We soon realized that to build effective systems, we need multiple specialized agents that can collaborate and self-organize.

To achieve this, several architectural patterns have emerged for AI agents.

This visual explains the 7 core patterns of multi-agent orchestration, each suited for specific workflows:



1) Parallel

Each agent tackles a different subtask, like data extraction, web retrieval, and summarization, and their outputs merge into a single result.

Perfect for reducing latency in high-throughput pipelines like document parsing or API orchestration.

2) Sequential

Each agent adds value step-by-step, like one generates code, another reviews it, and a third deploys it.

You'll see this in workflow automation, ETL chains, and multi-step reasoning pipelines.

3) Loop

Agents continuously refine their own outputs until a desired quality is reached.

Great for proofreading, report generation, or creative iteration, where the system thinks again before finalizing results.

4) Router

Here, a controller agent routes tasks to the right specialist. For instance, user queries about finance go to a FinAgent, legal queries to a LawAgent.

It's the foundation of context-aware agent routing, as seen in emerging MCP/A2A-style frameworks.

5) Aggregator

Many agents produce partial results that the main agent combines into one final output. So each agent forms an opinion, and a central one aggregates them into a consensus.

Common in RAG retrieval fusion, voting systems, etc.

6) Network

There's no clear hierarchy here, and agents just talk to each other freely, sharing context dynamically.

Used in simulations, multi-agent games, and collective reasoning systems where free-form behavior is desired.

7) Hierarchical

A top-level planner agent delegates subtasks to workers, tracks their progress,

and makes final calls. This is exactly like a manager and their team.

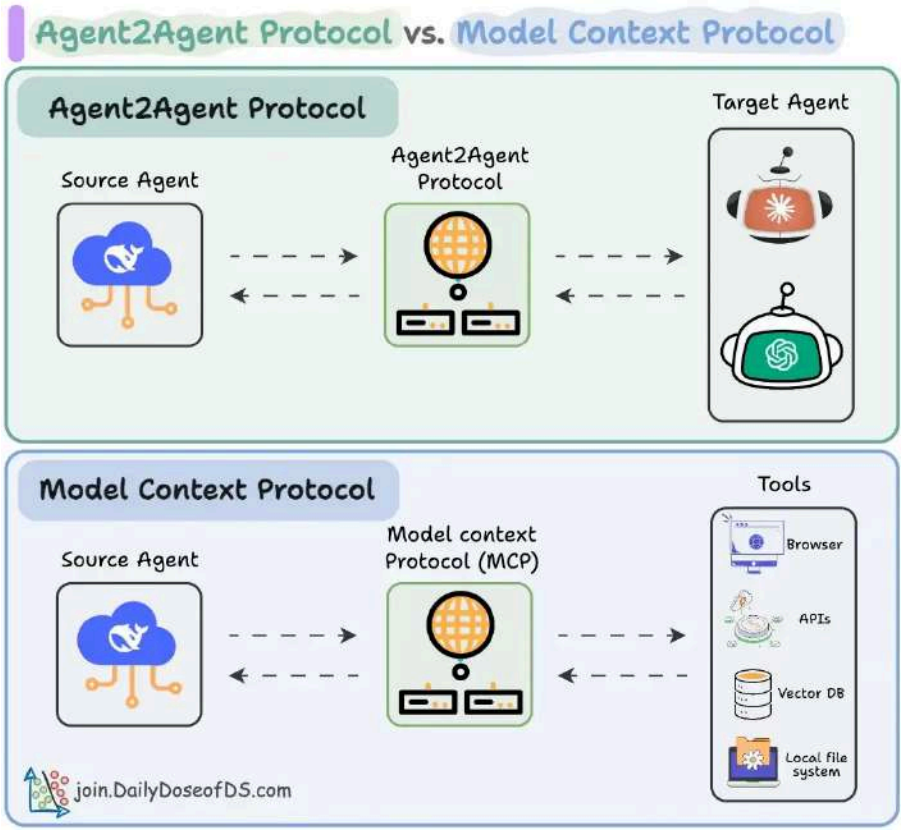
One thing we constantly think about when picking a pattern to build a multi-agent system (provided we do need an Agent and it has to be a multi-agent system) is not which one looks coolest, but which one minimizes friction between agents.

It's easy to spin up 10 agents and call it a team. What's hard is designing the communication flow so that:

- No two agents duplicate work.
- Every agent knows when to act and when to wait.
- The system collectively feels smarter than any individual part.

Agent2Agent(A2A) Protocol

Agentic applications require both A2A and MCP.



MCP provides agents with access to tools.

While A2A allows agents to connect with other agents and collaborate in teams.

Let's clearly understand what A2A is and how it can work with MCP.

If you don't know about MCP servers, we covered them in detail in the next section.

In a gist:

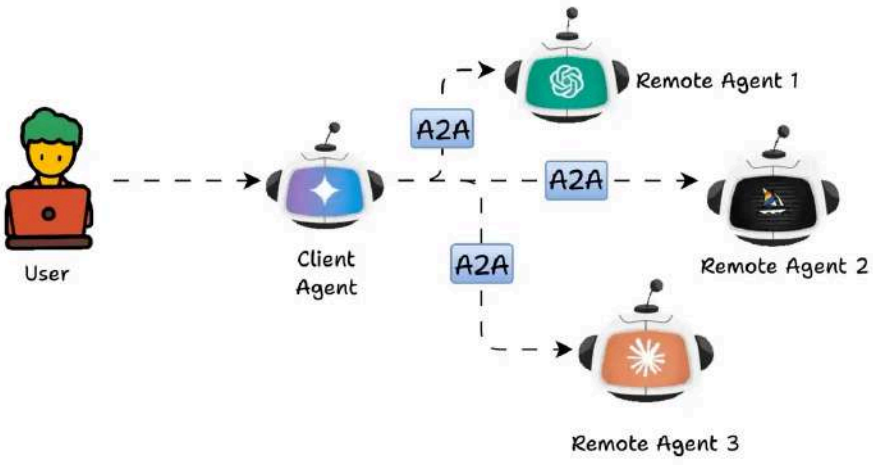
Agent2Agent (A2A) protocol lets AI agents connect to other Agents.

Model context protocol lets AI Agents connect to Tools/APIs.

So using A2A, while two Agents might be talking to each other...they themselves might be communicating to MCP servers.

In that sense, they do not compete with each other.

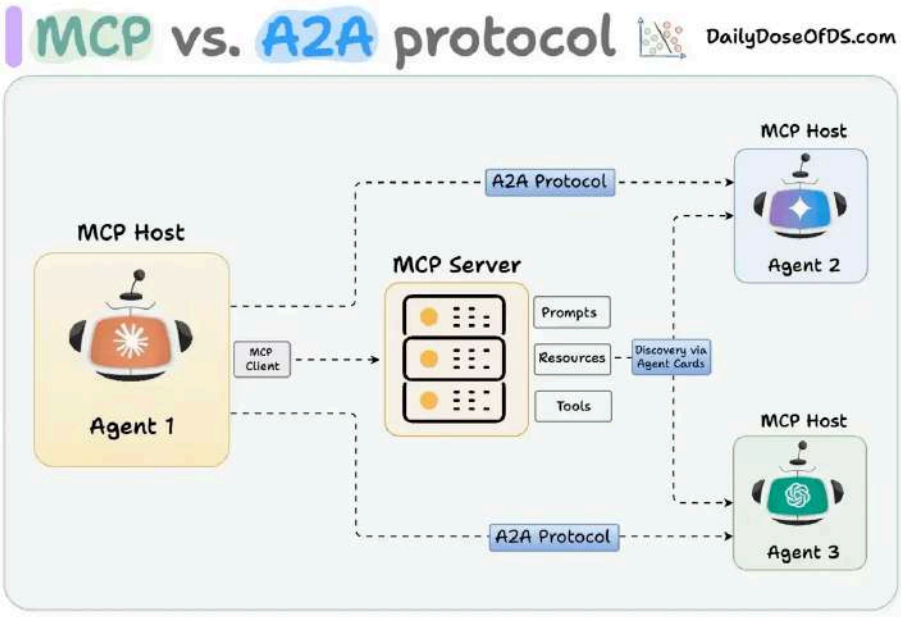
To explain further, Agent2Agent (A2A) enables multiple AI agents to work together on tasks without directly sharing their internal memory, thoughts, or tools.



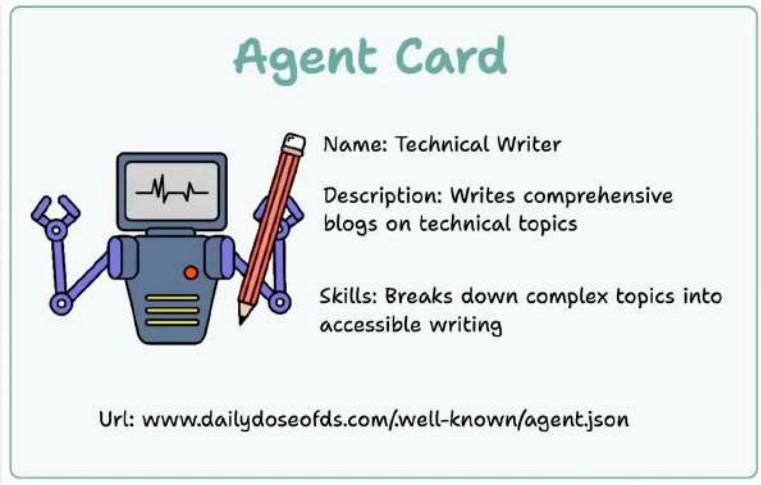
Instead, they communicate by exchanging context, task updates, instructions, and data.

Essentially, AI applications can model A2A agents as MCP resources,

represented by their AgentCard (more about it shortly).



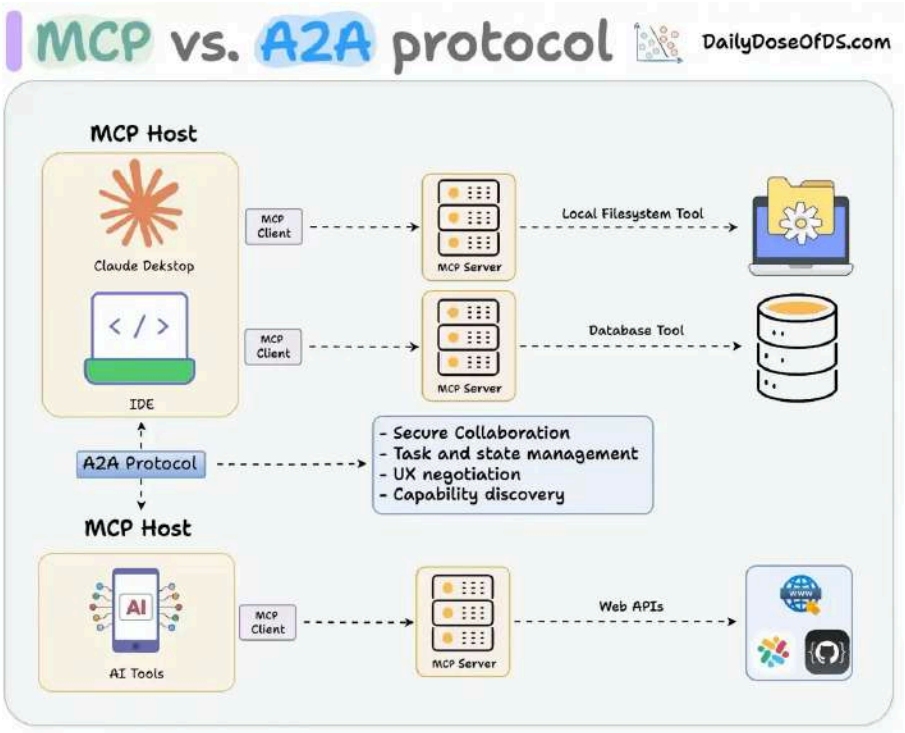
Using this, AI agents connecting to an MCP server can discover new agents to collaborate with and connect via the A2A protocol.



A2A-supporting Remote Agents must publish a "JSON Agent Card" detailing their capabilities and authentication.

Clients use this to find and communicate with the best agent for a task.

There are several things that make A2A powerful:



- Secure collaboration
- Task and state management
- Capability discovery
- Agents from different frameworks working together (LlamaIndex, CrewAI, etc.)

Additionally, it can integrate with MCP.

It's good to standardize Agent-to-Agent collaboration, similar to how MCP does for Agent-to-tool interaction.

Agent-User Interaction Protocol(AG-UI)

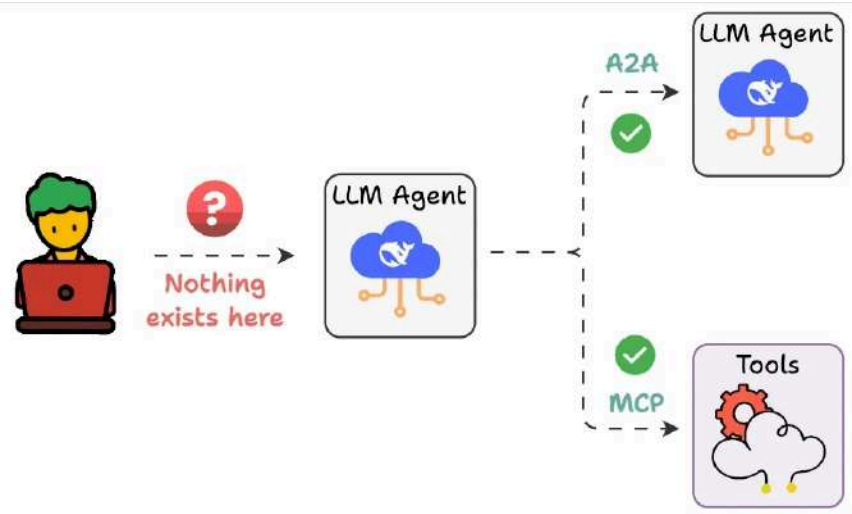
In the realm of Agents:

- MCP standardized Agent-to-Tool communication.

- Agent2Agent protocol standardized Agent-to-Agent communication.

But there's one piece still missing...

And that's a protocol for Agent-to-User communication:



Let's understand why this is important.

The problem

Today, you can build powerful multi-step agentic workflows using a toolkit like LangGraph, CrewAI, Mastra, etc.

But the moment you try to bring that Agent into a real-world app, things fall apart:

- You want to stream LLM responses token by token, without building a custom WebSocket server.

- You want to display tool execution progress as it happens, pause for human feedback, without blocking or losing context.
- You want to sync large, changing objects (like code or tables) without re-sending everything to the UI.
- You want to let users interrupt, cancel, or reply mid-agent run, without losing context.

And here's another issue:

Every Agent backend has its own mechanisms for tool calling, ReAct-style planning, state diffs, and output formats.

So if you use LangGraph, the front-end will implement custom WebSocket logic, messy JSON formats, and UI adapters specific to LangGraph.

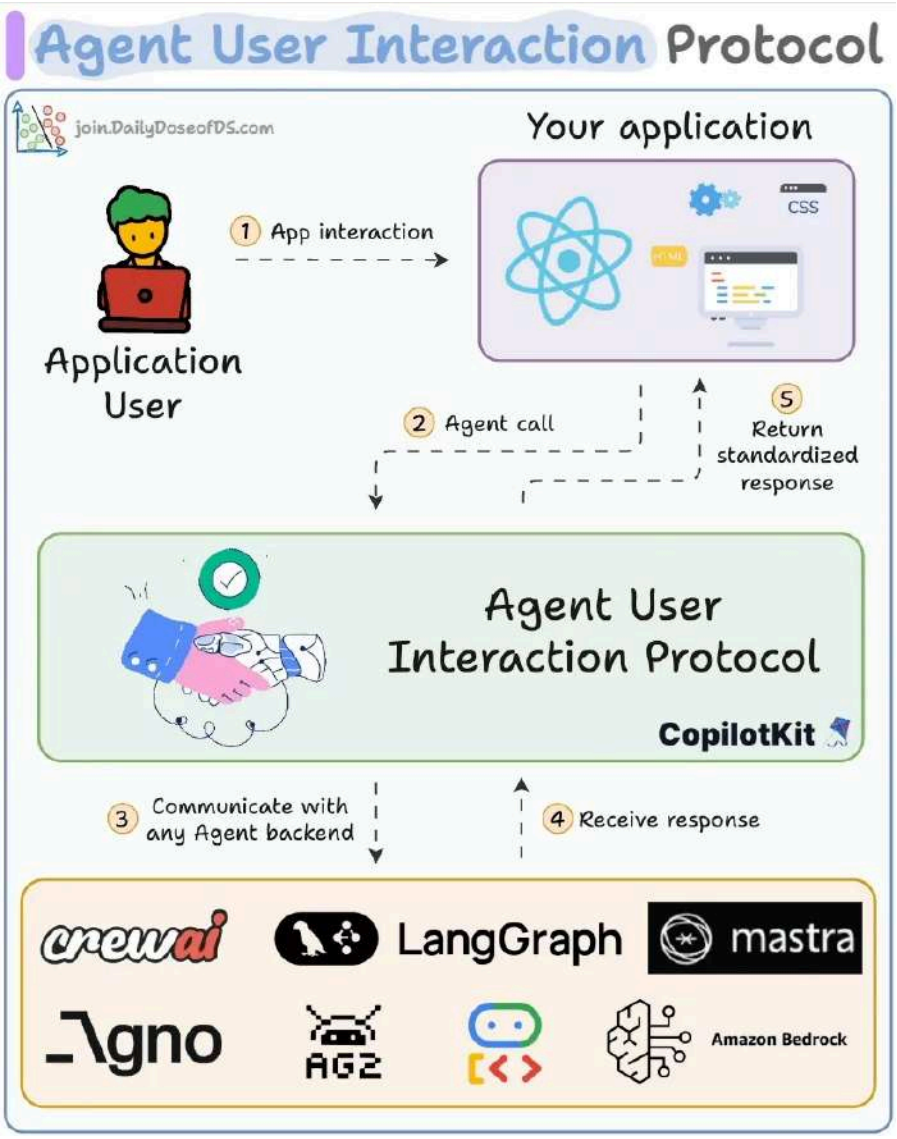
But to migrate to CrewAI, everything must be adapted.

This doesn't scale.

The solution: AG-UI

AG-UI (Agent-User Interaction Protocol) is an open-source protocol by CopilotKit that solves this.

It standardizes the interaction layer between backend agents and frontend UIs (the green layer below).



Think of it this way:

- Just like REST is the standard for client-to-server requests...
- AG-UI is the standard for streaming real-time agent updates back to the UI.

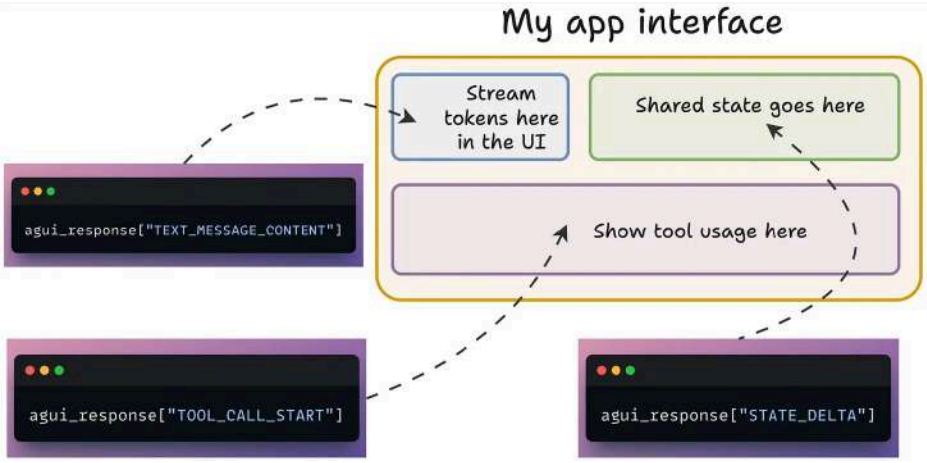
Technically speaking...

It uses Server-Sent Events (SSE) to stream structured JSON events to the frontend.

Each event has an explicit payload (like keys in a Python dictionary) like:

- TEXT_MESSAGE_CONTENT for token streaming.
- TOOL_CALL_START to show tool execution.
- STATE_DELTA to update shared state (code, data, etc.)
- AGENT_HANDOFF to smoothly pass control between agents

And it comes with SDKs in TypeScript and Python to make this plug-and-play for any stack, like shown below:



In the above image, the response from the Agent is not specific to any toolkit. It is a standardized AG-UI response.

This means you need to write your backend logic once and hook it into AG-UI, and everything just works:

- LangGraph, CrewAI, Mastra—all can emit AG-UI events.
- UIs can be built using CopilotKit components or your own React stack.
- You can swap GPT-4 for Llama-3 locally and not change anything in the frontend.

This is the layer that will make your Agent apps feel like real software, not just glorified chatbots.

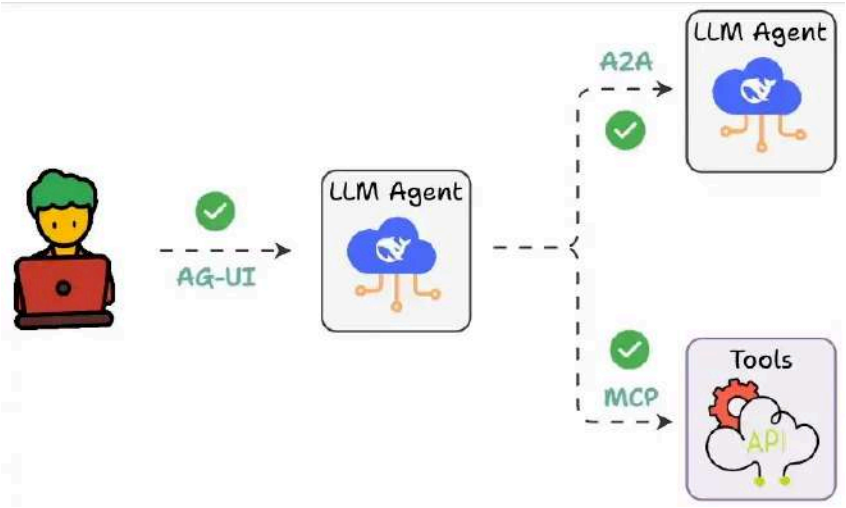
Agent Protocol Landscape

Something remarkable is happening in the AI industry.

Earlier the agent ecosystem was fragmented into dozens of incompatible frameworks.

But finally, the industry is converging around three protocols that work together.

These are:



AG-UI (Agent-User Interaction):

- The bi-directional connection between agentic backends and frontends.
- This is how agents become truly interactive inside your apps, not just as chatbots, but collaborative co-workers.

MCP (Model Context Protocol):

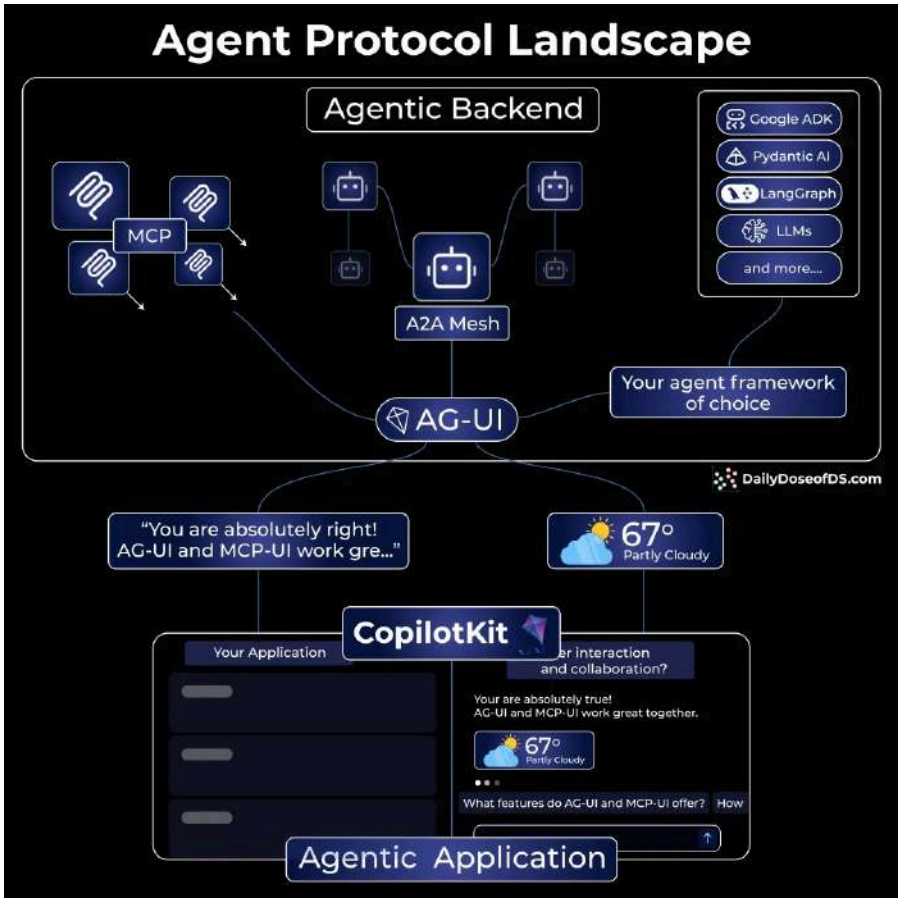
- The standard for how agents connect to tools, data, and workflows.
- Started by Anthropic, now adopted everywhere.

A2A (Agent-to-Agent):

- The protocol for multi-agent coordination.
- How agents delegate tasks and share intent across systems.

These aren't competing standards. They're layers of the same stack.

AG-UI can handshake with both MCP and A2A, meaning tool outputs and multi-agent collaboration can flow seamlessly to your user interface.



Your frontend stays connected to the entire agent ecosystem through one unified protocol layer.

CopilotKit sits above all three as the Agentic Application Framework.

It acts as the practical layer that lets you actually build with these protocols without dealing with the complexity.

So you get all three protocols, generative UI support and production-ready infrastructure in one framework.

The best part: All of this is 100% open-source!

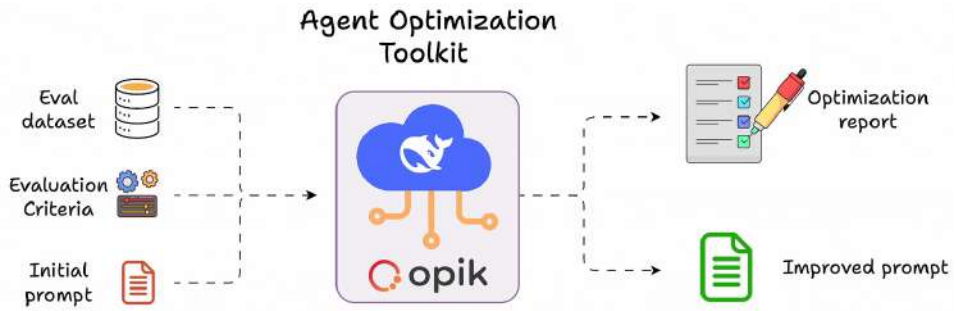
It breaks down handshakes, misconceptions and real examples and shows exactly how to start building.

Agent optimization with Opik

Developers manually iterate through prompts to find an optimal one. This is not scalable and performance can degrade across models.

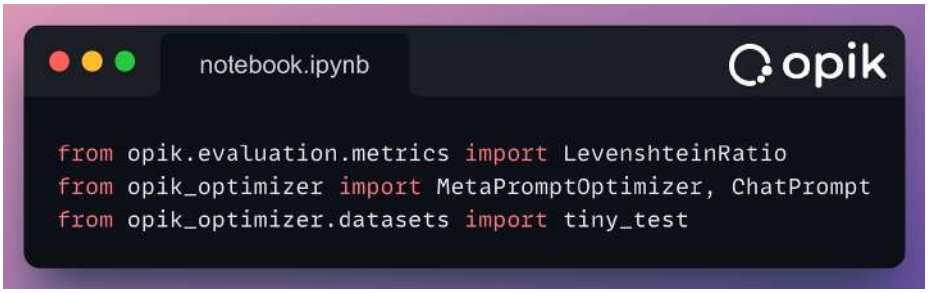
Let's learn how to use the Opik Agent Optimizer toolkit that lets you automatically optimize prompts for LLM apps.

The idea is to start with an initial prompt and an evaluation dataset, and let an LLM iteratively improve the prompt based on evaluations.



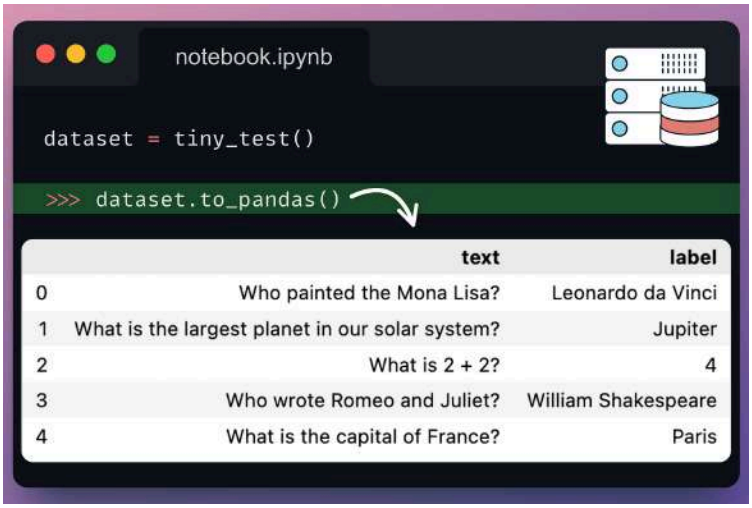
To begin, install Opik and its optimizer package, and configure Opik:

Next, import all the required classes and functions from opik and opik_optimizer:



- LevenshteinRatio → Our metric to evaluate the prompt's effectiveness in generating a precise output for the given input.
- MetaPromptOptimizer → An algorithm that uses a reasoning model to critique and iteratively refine your initial instruction prompt.
- tiny_test → A basic test dataset with input-output pairs.

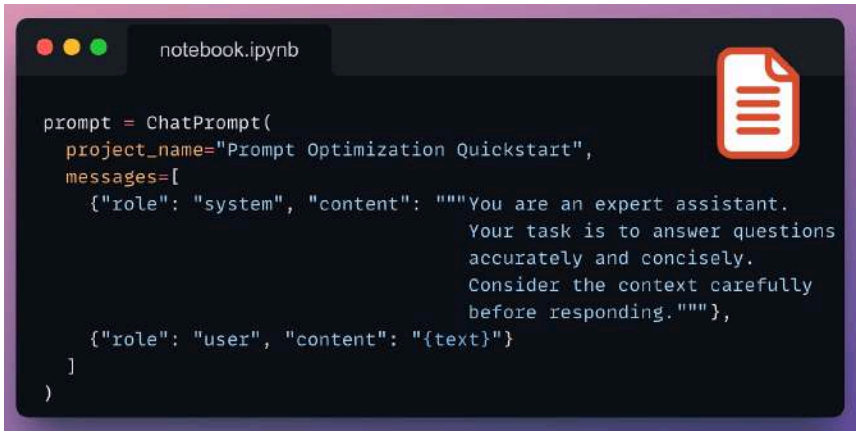
Next, define an evaluation dataset:



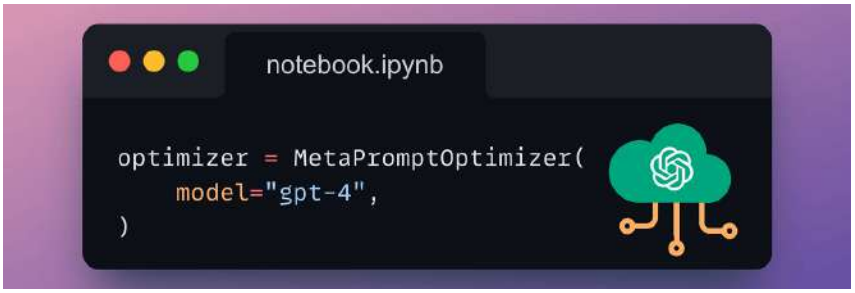
	text	label
0	Who painted the Mona Lisa?	Leonardo da Vinci
1	What is the largest planet in our solar system?	Jupiter
2	What is 2 + 2?	4
3	Who wrote Romeo and Juliet?	William Shakespeare
4	What is the capital of France?	Paris

Moving on, configure the evaluation metric, which tells the optimizer how to score the LLM's outputs against the given label:

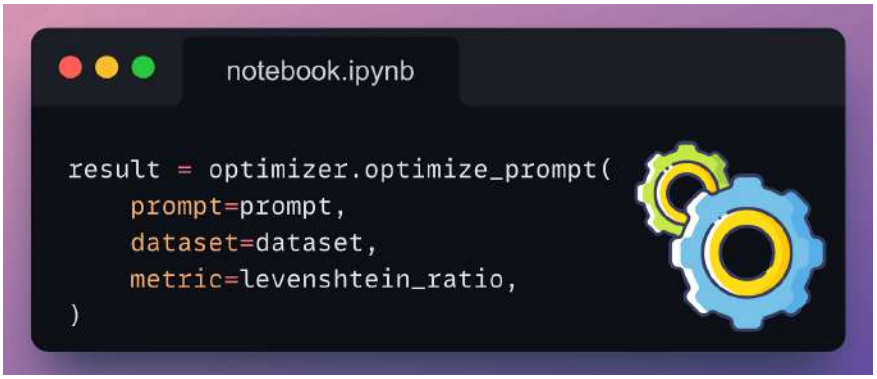
Next, define your base prompt, which is the initial instruction that the MetaPromptOptimizer will try to enhance:



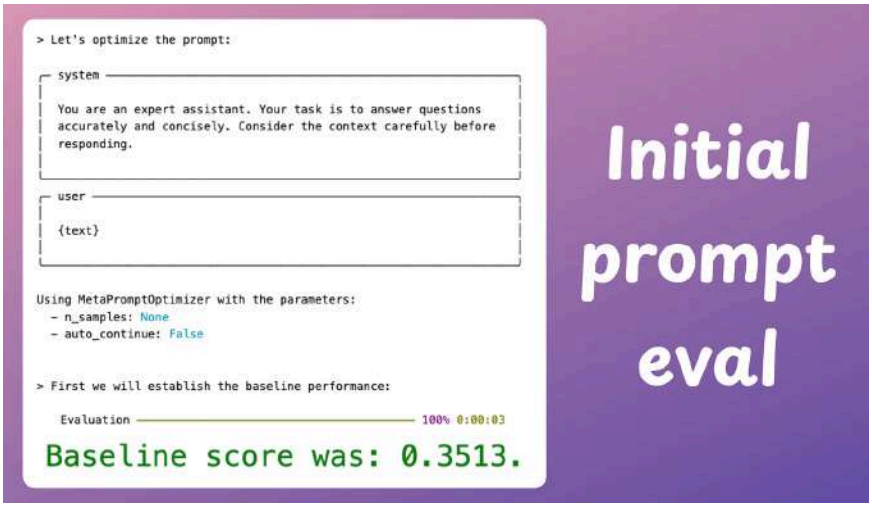
Next, instantiate a MetaPromptOptimizer, specifying the model to use in the optimization process:



Finally, the optimizer.optimize_prompt(...) method is invoked with the dataset, metric configuration, and prompt to start the optimization process:

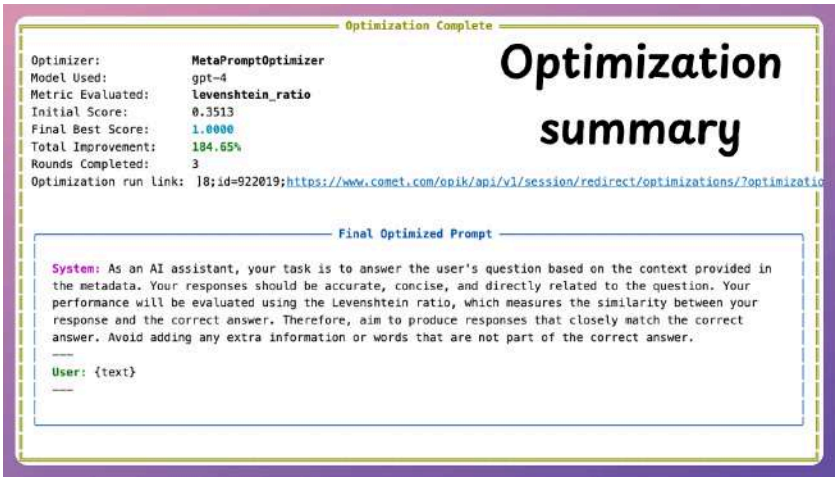


It starts by evaluating the initial prompt, which sets the baseline:

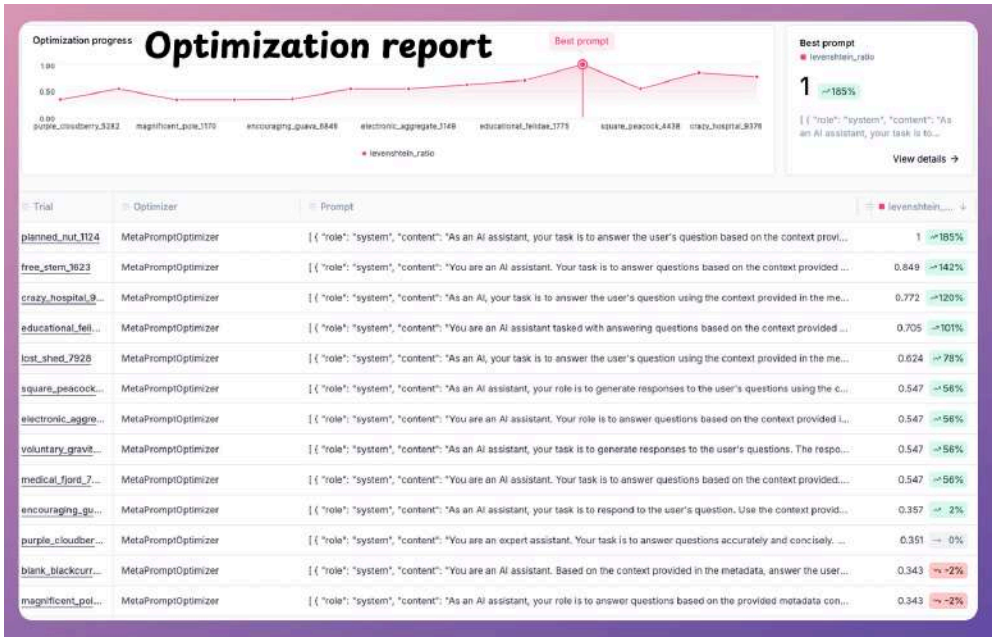


Then it iterates through several different prompts (written by AI), evaluates them,

and prints the most optimal prompt. You can invoke result.display() to see a summary of the optimization, the best prompt found and its score:



The optimization results are also available in the Opik dashboard for further analysis and visualization:



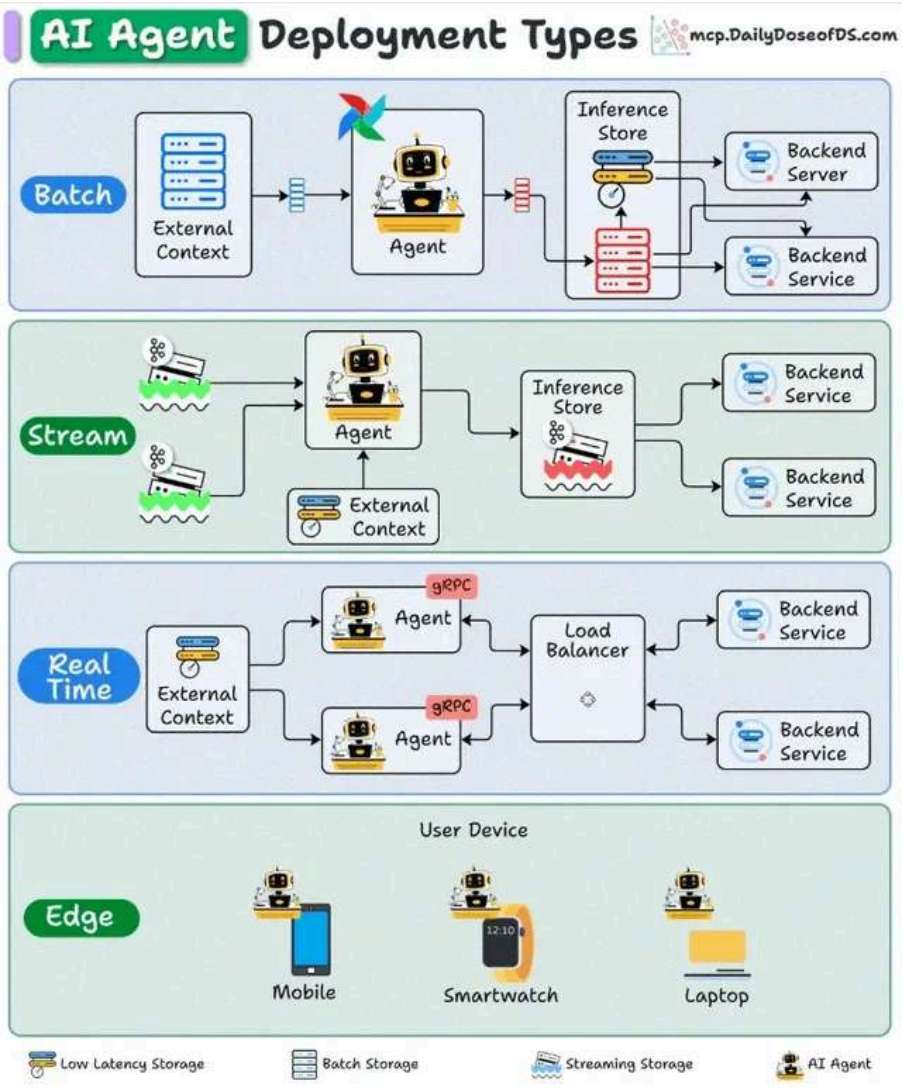
And that's how you can use Opik Agent Optimizer to enhance the performance and efficiency of your LLM apps.

Note: While we used GPT-4o, everything here can be executed 100% locally since you can use any other LLM + Opik is fully open-source.

AI Agent Deployment Strategies

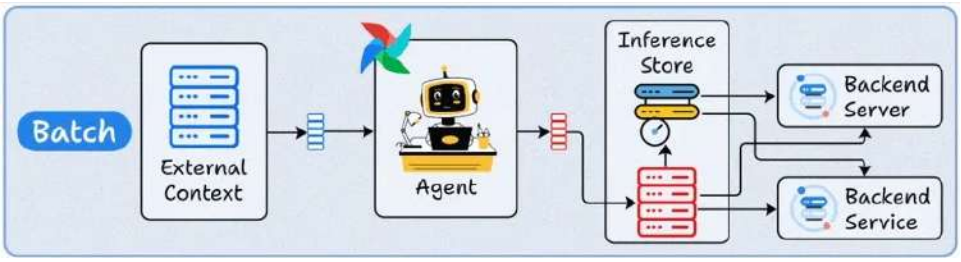
Deploying AI agents isn't one-size-fits-all. The architecture you choose can make or break your agent's performance, cost efficiency, and user experience.

Here are the 4 main deployment patterns you need to know:



1) Batch deployment

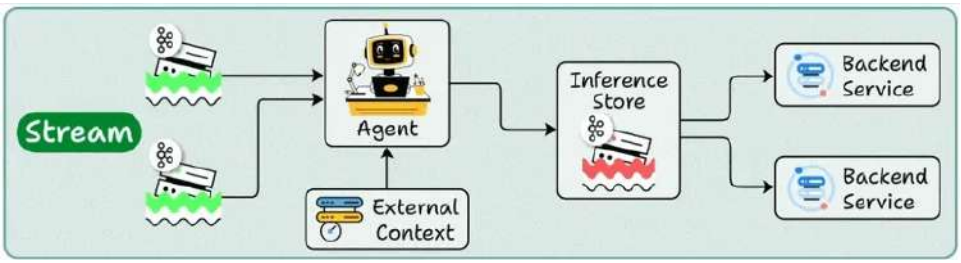
You can think of this as a scheduled automation.



- The Agent runs periodically, like a scheduled CLI job.
- Just like any other Agent, it can connect to external context (databases, APIs, or tools), process data in bulk, and store results.
- This typically optimizes for throughput over latency.
- This is best for processing large volumes of data that don't need immediate responses.

2) Stream deployment

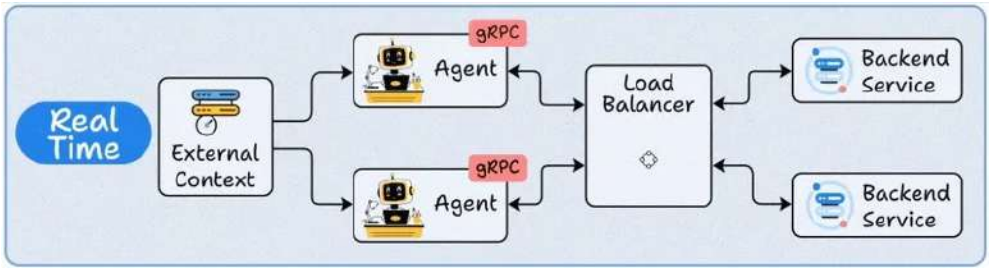
Here, the Agent becomes part of a streaming data pipeline.



- It continuously processes data as it flows through systems.
- Your agent stays active, handling concurrent streams while accessing both streaming storage and backend services as needed.
- Multiple downstream applications can then make use of these processed outputs.
- Best for: Continuous data processing and real-time monitoring

3) Real-Time deployment

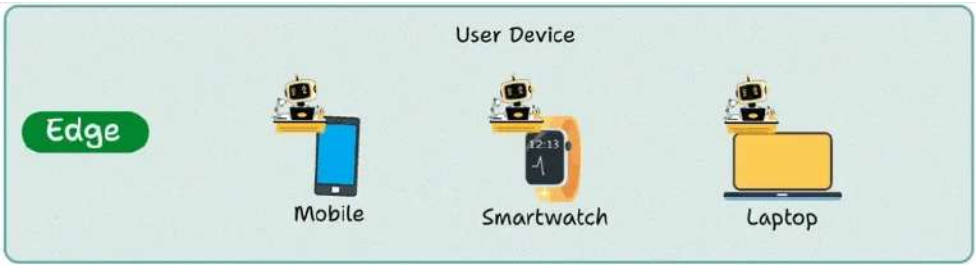
This is where Agents act like live backend services.



- The Agent runs behind an API (REST or gRPC).
- When a request arrives, it retrieves any needed context, reasons using the LLM, and responds instantly.
- Load balancers ensure scalability across multiple concurrent requests.
- This is your go-to for chatbots, virtual assistants, and any application where users expect sub-second responses.

4) Edge deployment

The agent runs directly on user devices: mobile phones, smartwatches, and laptops so no server round-trip is needed.



- The reasoning logic lives inside your mobile, smartwatch, or laptop.
- Sensitive data never leaves the device, improving privacy and security.
- Useful for tasks that need to work offline or maintain user confidentiality.
- Best for: Privacy-first applications and offline functionality

To summarize:

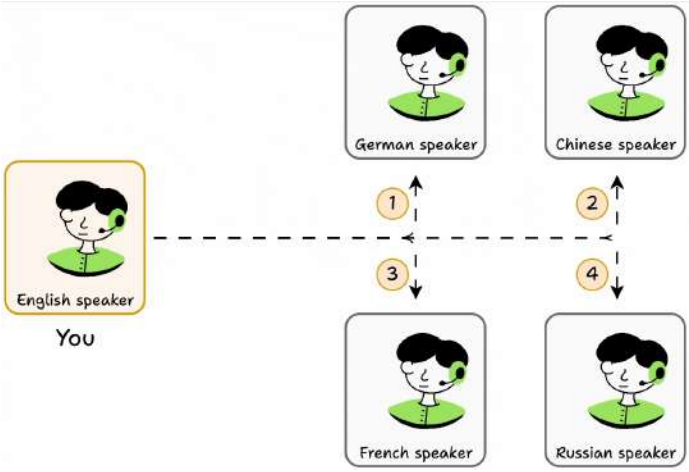
- Batch = Maximum throughput
- Stream = Continuous processing
- Real-Time = Instant interaction
- Edge = Privacy + offline capability

Each pattern serves different needs. The key is matching your deployment strategy to your specific use case, performance requirements, and user expectations.

MCP

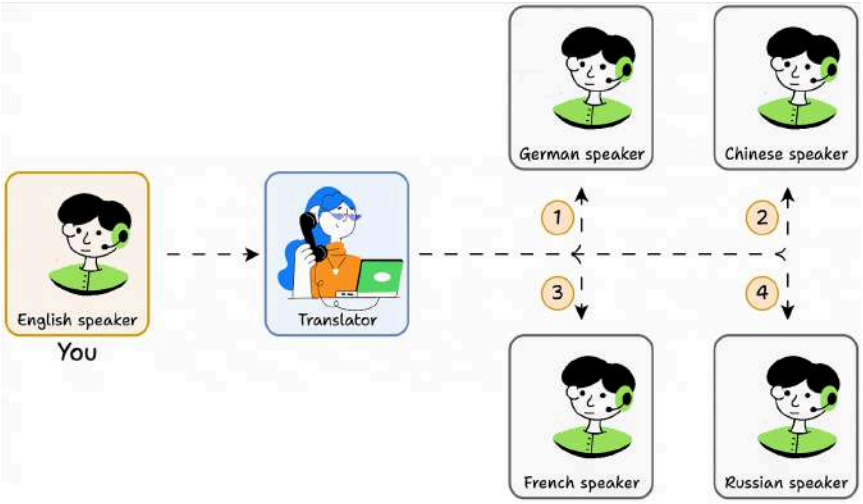
What is MCP?

Imagine you only know English. To get info from a person who only knows:



- French, you must learn French.
- German, you must learn German.
- And so on.

In this setup, learning even 5 languages will be a nightmare for you.
But what if you add a translator that understands all languages?

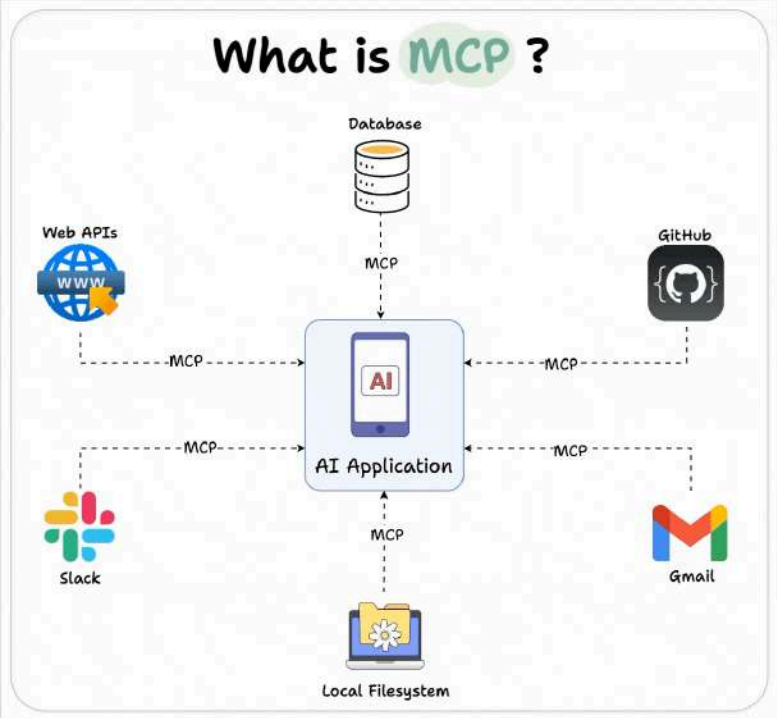


This is simple, isn't it?

The translator is like an MCP!

It lets you (Agents) talk to other people (tools or other capabilities) through a single interface.

To formalize, while LLMs possess impressive knowledge and reasoning skills, which allow them to perform many complex tasks, their knowledge is limited to their initial training data.



If they need to access real-time information, they must use external tools and resources on their own.

Model context protocol (MCP) is a standardized interface and framework that allows AI models to seamlessly interact with external tools, resources, and environments.

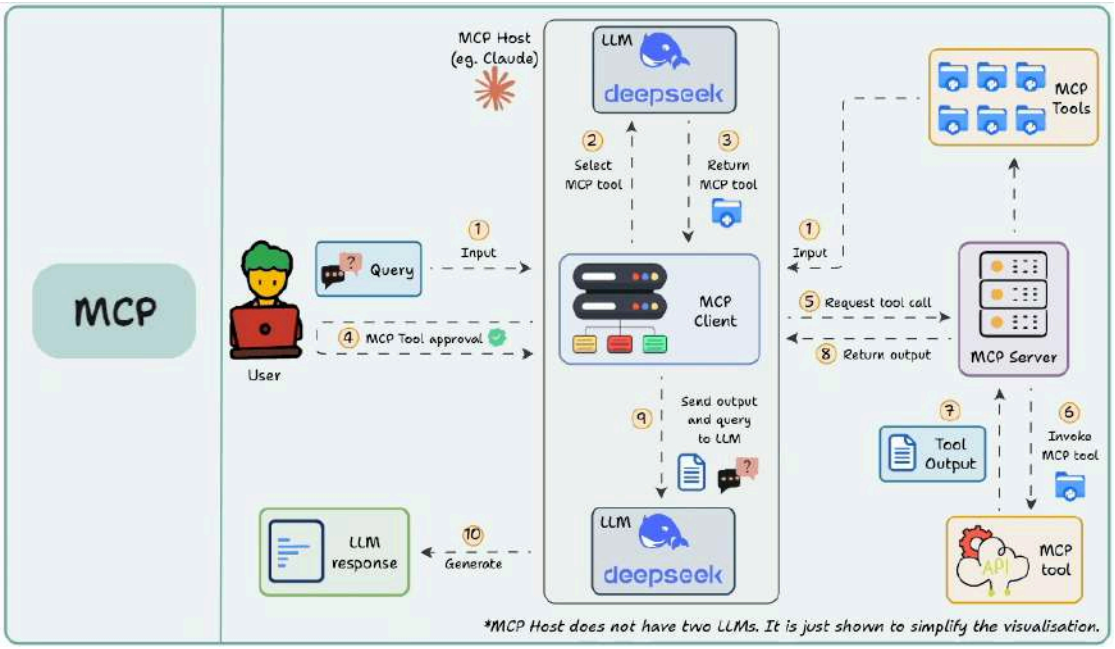
MCP acts as a universal connector for AI systems to capabilities (tools, etc.), similar to how USB-C standardizes connections between electronic devices.

Why was MCP created?

Without MCP, adding a new tool or integrating a new model was a headache.

If you had three AI applications and three external tools, you might end up writing nine different integration modules (each AI x each tool) because there was no common standard. This doesn't scale.

Developers of AI apps were essentially reinventing the wheel each time, and tool providers had to support multiple incompatible APIs to reach different AI platforms.



Let's understand this in detail.

The problem

Before MCP, the landscape of connecting AI to external data and actions looked like a patchwork of one-off solutions.

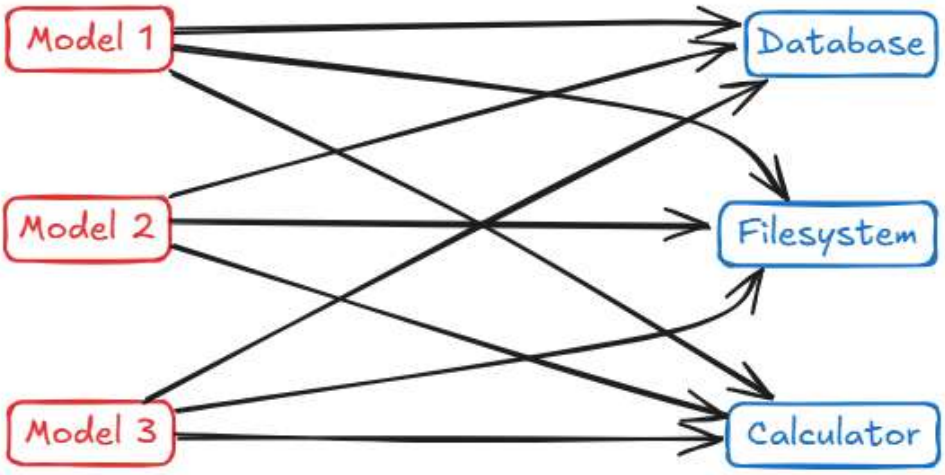
Either you hard-coded logic for each tool, managed prompt chains that were not

robust, or you used vendor-specific plugin frameworks.

This led to the infamous M×N integration problem.

Essentially, if you have M different AI applications and N different tools/data sources, you could end up needing M × N custom integrations.

The diagram below illustrates this complexity: each AI (each "Model") might require unique code to connect to each external service (database, filesystem, calculator, etc.), leading to spaghetti-like interconnections.

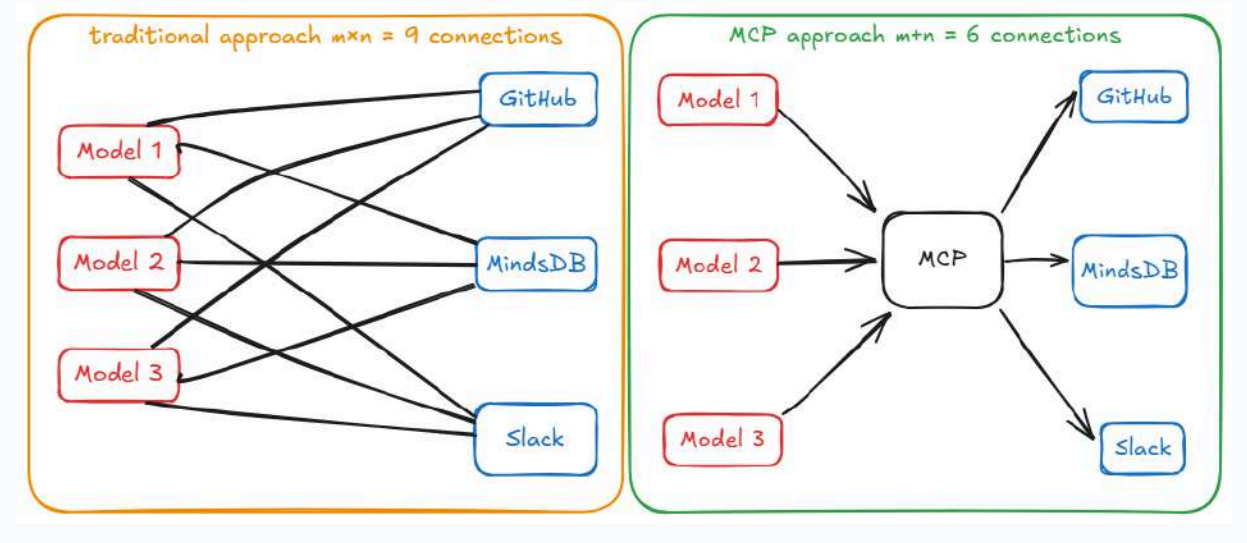


The solution

MCP tackles this by introducing a standard interface in the middle. Instead of M × N direct integrations, we get M + N implementations: each of the M AI applications implements the MCP client side once, and each of the N tools implements an MCP server once.

Now everyone speaks the same "language", so to speak, and a new pairing doesn't require custom code since they already understand each other via MCP.

The following diagram illustrates this shift.

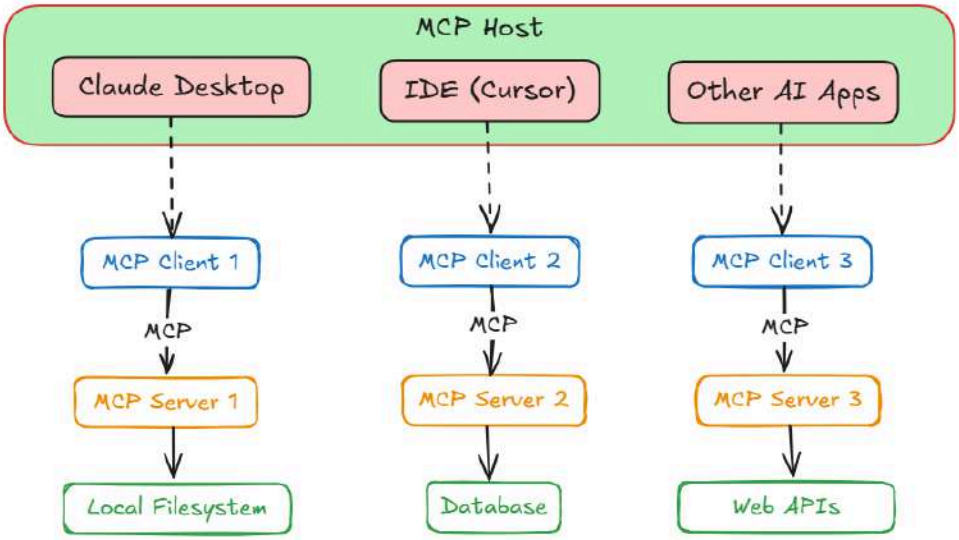


- On the left (pre-MCP), every model had to wire into every tool.
- On the right (with MCP), each model and tool connects to the MCP layer, drastically simplifying connections. You can also relate this to the translator example we discussed earlier.

MCP Architecture Overview

At its heart, MCP follows a client-server architecture (much like the web or other network protocols).

However, the terminology is tailored to the AI context. There are three main roles to understand: the Host, the Client, and the Server.



Host

The Host is the user-facing AI application, the environment where the AI model lives and interacts with the user.

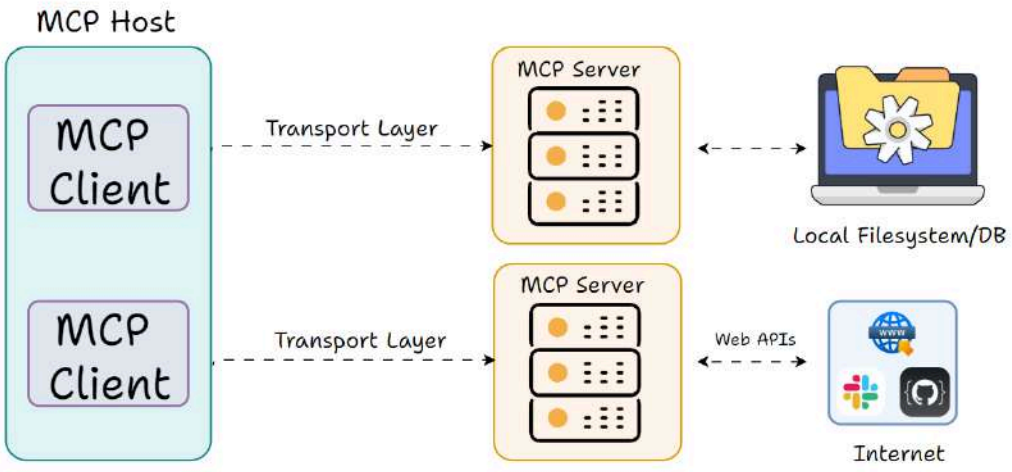
This could be a chat application (like OpenAI's ChatGPT interface or Anthropic's Claude desktop app), an AI-enhanced IDE (like Cursor), or any custom app that embeds an AI assistant like Chainlit.

Host is the one that initiates connections to the available MCP servers when the system needs them. It captures the user's input, keeps the conversation history, and displays the model's replies.

Client

The MCP Client is a component within the Host that handles the low-level communication with an MCP Server.

Think of the Client as the adapter or messenger. While the Host decides what to do, the Client knows how to speak MCP to actually carry out those instructions with the server.

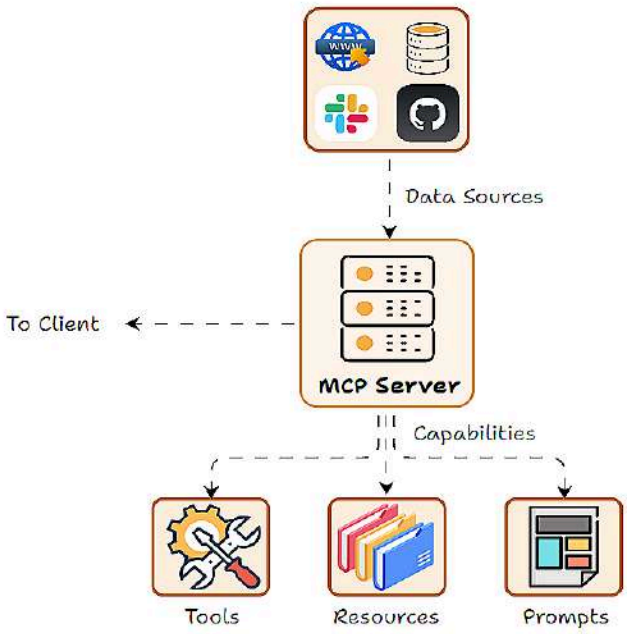


Server

The MCP Server is the external program or service that actually provides the capabilities (tools, data, etc.) to the application.

An MCP Server can be thought of as a wrapper around some functionality, which exposes a set of actions or resources in a standardized way so that any MCP Client can invoke them.

Servers can run locally on the same machine as the Host or remotely on some cloud service since MCP is designed to support both scenarios seamlessly. The key is that the Server advertises what it can do in a standard format (so the client can query and understand available tools) and will execute requests coming from the client, then return results.



Tools, Resources and Prompts

Tools, prompts and resources form the three core capabilities of the MCP framework. Capabilities are essentially the features or functions that the server makes available.

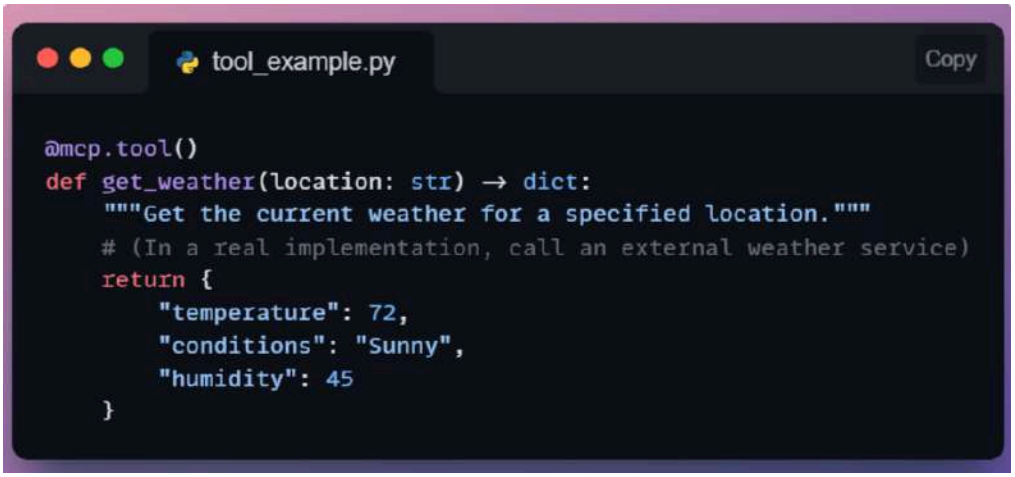
- Tools: Executable actions or functions that the AI (host/client) can invoke (often with side effects or external API calls).
- Resources: Read-only data sources that the AI (host/client) can query for information (no side effects, just retrieval).
- Prompts: Predefined prompt templates or workflows that the server can supply.

Tools

Tools are what they sound like: functions that do something on behalf of the AI model. These are typically operations that can have effects or require computation beyond the AI's own capabilities.

Importantly, tools are usually triggered by the AI model's choice, which means the LLM (via the host) decides to call a tool when it determines it needs that functionality.

Suppose we have a simple tool for weather. In an MCP server's code, it might look like:

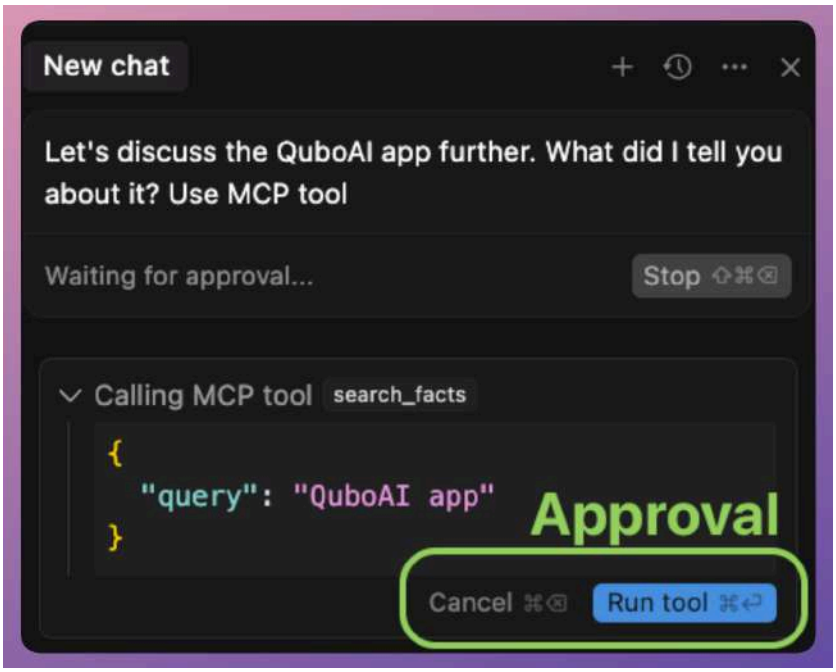


This Python function, registered with @mcp.tool(), can be invoked by the AI via MCP.

When the AI calls tools/call with name "get_weather" and {"location": "San Francisco"} as arguments, the server will execute get_weather("San Francisco") and return the dictionary result.

The client will get that JSON result and make it available to the AI. Notice the tool returns structured data (temperature, conditions), and the AI can then use or verbalize (generate a response) that info.

Since tools can do things like file I/O or network calls, an MCP implementation often requires that the user permit a tool call.



For example, Claude's client might pop up "The AI wants to use the 'get_weather' tool, allow yes/no?" the first time, to avoid abuse. This ensures the human stays in control of powerful actions.

Tools are analogous to "functions" in classic function calling, but under MCP, they are used in a more flexible, dynamic context. They are model-controlled but developer/governance-approved in execution.

Resources

Resources provide read-only data to the AI model.

These are like databases or knowledge bases that the AI can query to get information, but not modify.

Unlike tools, resources typically do not involve heavy computation or side effects, since they are often just information lookups.

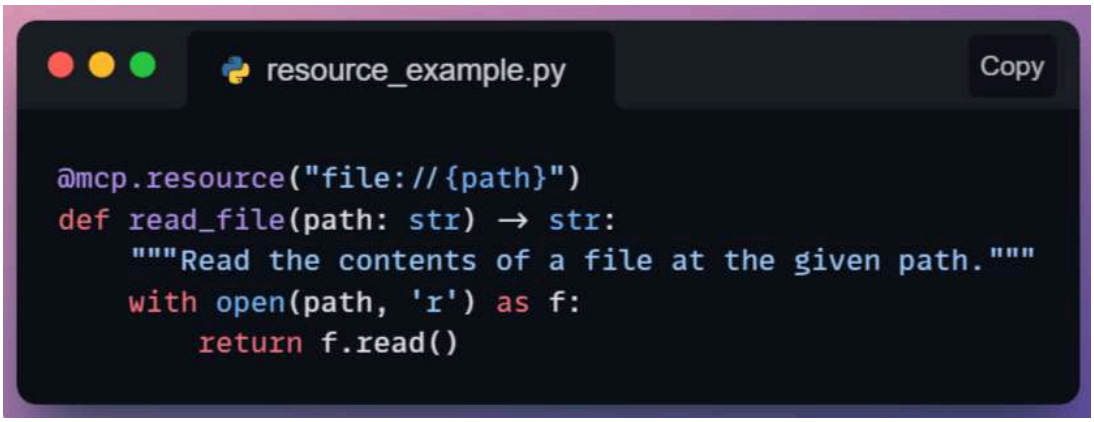
Another key difference is that resources are usually accessed under the host application's control (not spontaneously by the model). In practice, this might mean the Host knows when to fetch a certain context for the model.

For instance, if a user says, "Use the company handbook to answer my question," the Host might call a resource that retrieves relevant handbook sections and feeds them to the model.

Resources could include a local file's contents, a snippet from a knowledge base or documentation, a database query result (read-only), or any static data like configuration info.

Essentially anything the AI might need to know as context. An AI research assistant could have resources like "ArXiv papers database," where it can retrieve an abstract or reference when asked.

A simple resource could be a function to read a file:



Here we use a decorator @mcp.resource("file://{path}") which might indicate a template for resource URIs.

The AI (or Host) could ask the server for resources.get with a URI like file://home/user/notes.txt, and the server would callread_file("/home/user/notes.txt") and return the text.

Notice that resources are usually identified by some identifier (like a URI or name) rather than being free-form functions.

They are also often application-controlled, meaning the app decides when to retrieve them (to avoid the model just reading everything arbitrarily).

From a safety standpoint, since resources are read-only, they are less dangerous, but still, one must consider privacy and permissions (the AI shouldn't read files it's not supposed to).

The Host can regulate which resource URIs it allows the AI to access, or the server might restrict access to certain data.

In summary, Resources give the AI knowledge without handing over the keys to change anything.

They're the MCP equivalent of giving the model reference material when needed, which acts like a smarter, on-demand retrieval system integrated through the protocol.

Prompts

Prompts in the MCP context are a special concept: they are predefined prompt templates or conversation flows that can be injected to guide the AI's behavior.

Essentially, a Prompt capability provides a canned set of instructions or an example dialogue that can help steer the model for certain tasks.

But why have prompts as a capability?

Think of recurring patterns: e.g., a prompt that sets up the system role as "You

are a code reviewer," and the user's code is inserted for analysis.

Rather than hardcoding that in the host application, the MCP server can supply it.

Prompts can also represent multi-turn workflows.

For instance, a prompt might define how to conduct a step-by-step diagnostic interview with a user. By exposing this via MCP, any client can retrieve and use these sophisticated prompts on demand.

As far as control is concerned, Prompts are usually user-controlled or developer-controlled.

The user might pick a prompt/template from a UI (e.g., "Summarize this document" template), which the host then fetches from the server.

The model doesn't spontaneously decide to use prompts the way it does tools.

Rather, the prompt sets the stage before the model starts generating. In that sense, prompts are often fetched at the beginning of an interaction or when the user chooses a specific "mode".

Suppose we have a prompt template for code review. The MCP server might have:

This prompt function returns a list of message objects (in OpenAI format) that set up a code review scenario.

When the host invokes this prompt, it gets those messages and can insert the actual code to be reviewed into the user content.

Then it provides these messages to the model before the model's own answer. Essentially, the server is helping to structure the conversation.

While we have personally not seen much applicability of this yet, common use cases for prompt capabilities include things like "brainstorming guide," "step-by-step problem solver template," or domain-specific system roles.

By having them on the server, they can be updated or improved without changing the client app, and different servers can offer different specialized prompts.

An important point to note here is that prompts, as a capability, blur the line between data and instructions.

They represent best practices or predefined strategies for the AI to use.

In a way, MCP prompts are similar to how ChatGPT plugins can suggest how to format a query, but here it's standardized and discoverable via the protocol.

API versus MCP

APIs (Application Programming Interfaces) and MCPs are both used for communication between software systems, but they have distinct purposes and are designed for different use cases.

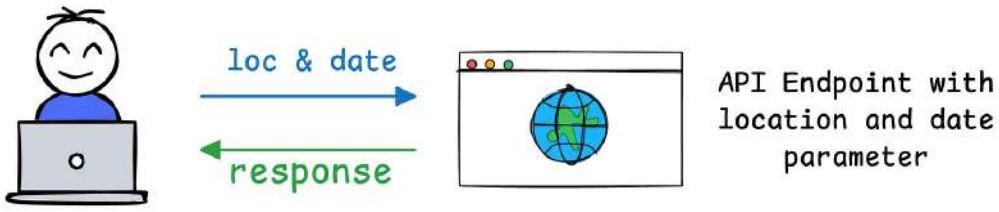
APIs are general-purpose interfaces for software-to-software communication, while MCPs are specifically designed for AI agents to interact with external tools and data.

Key differences:

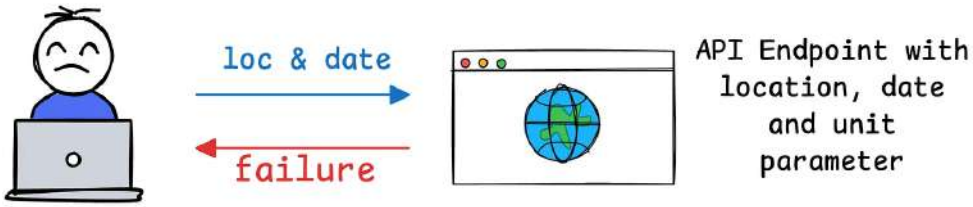
- MCPs aim to standardize how AI agents interact with tools, while APIs can vary greatly in their implementation.
- MCPs are designed to manage dynamic, evolving context, including data resources, executable tools, and prompts for workflows.
- MCPs are particularly well-suited for AI agents that need to adapt to new capabilities and tools without pre-programming.

In a traditional API setup:

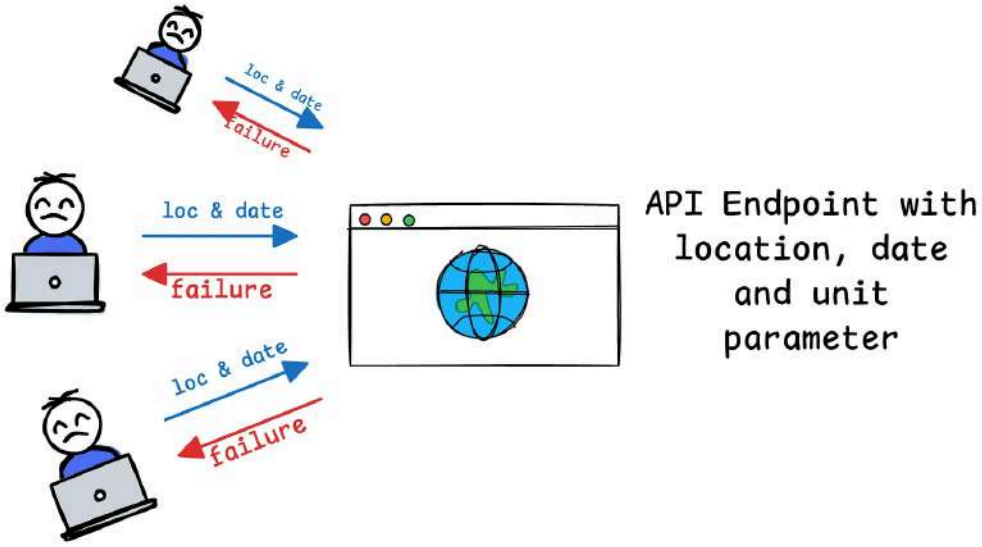
- If your API initially requires two parameters (e.g., location and date for a weather service), users integrate their applications to send requests with those exact parameters.



- Later, if you decide to add a third required parameter (e.g., unit for temperature units like Celsius or Fahrenheit), the API's contract changes.

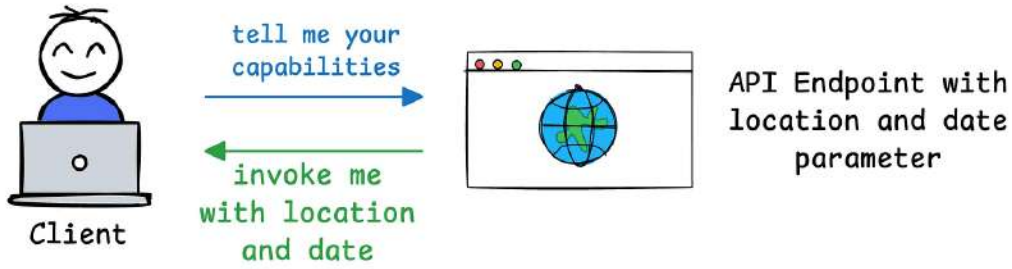


- This means all users of your API must update their code to include the new parameter. If they don't update, their requests might fail, return errors, or provide incomplete results.

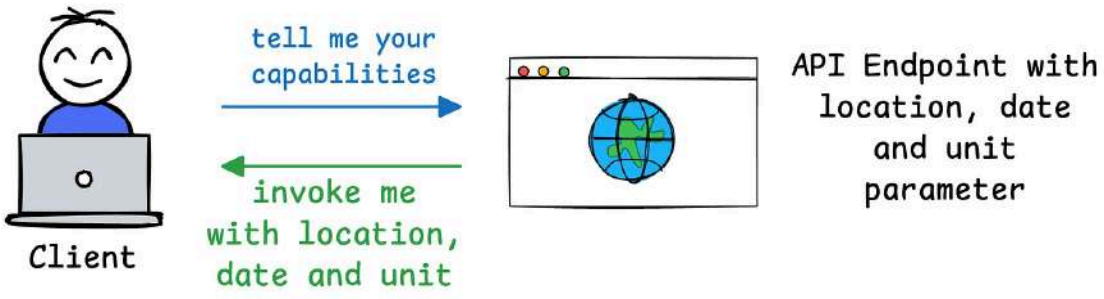


MCP's design solves this as follows:

- For instance, when a client (e.g., an AI application like Claude Desktop) connects to an MCP server (e.g., your weather service), it sends an initial request to learn the server's capabilities.
- The server responds with details about its available tools, resources, prompts, and parameters. For example, if your weather API initially supports *location* and *date*, the server communicates these as part of its capabilities.



- If you later add a *unit* parameter, the MCP server can dynamically update its capability description during the next exchange. The client doesn't need to hardcode or predefine the parameters since it simply queries the server's current capabilities and adapts accordingly.



- This way, the client can then adjust its behavior on-the-fly, using the updated capabilities (e.g., including unit in its requests) without needing to rewrite or redeploy code.

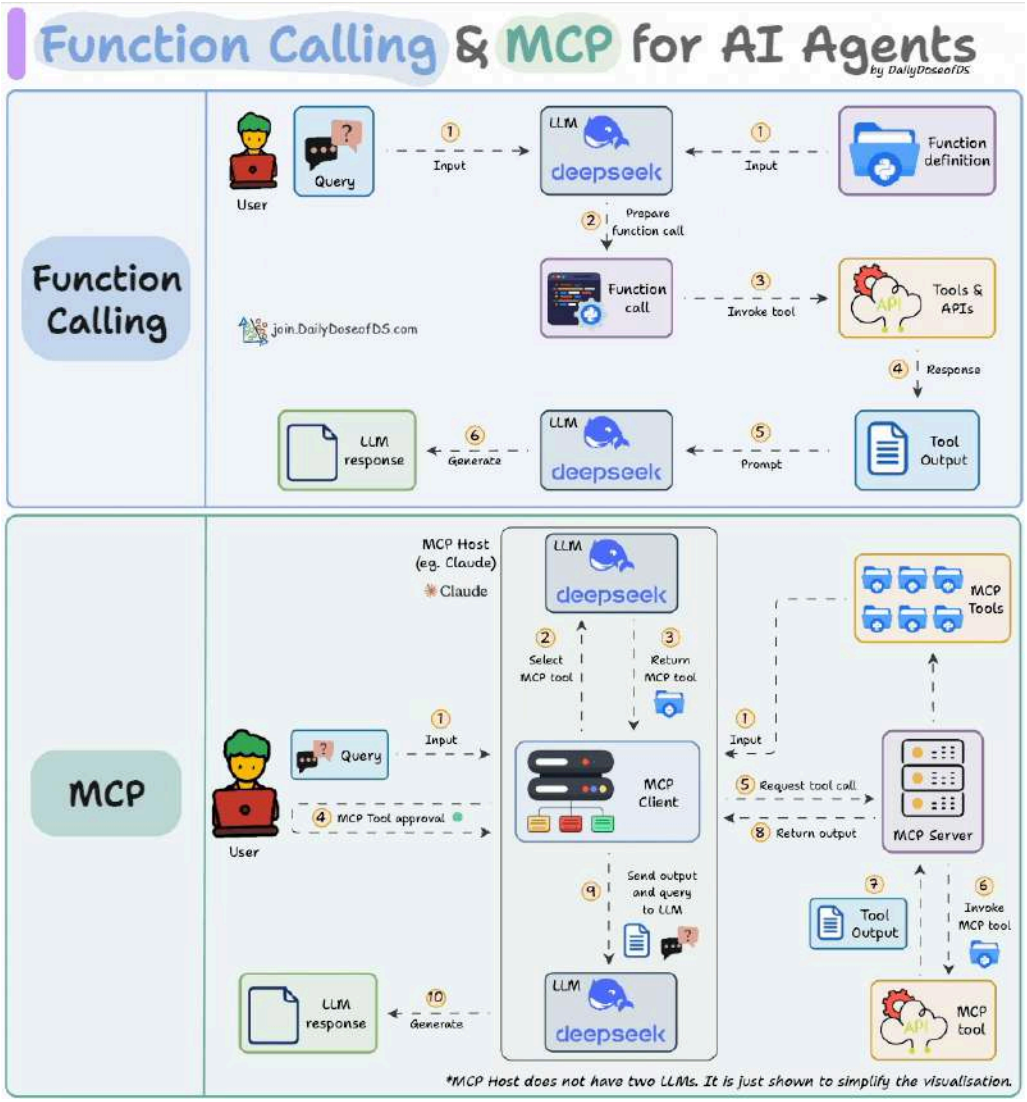
We'll understand this topic better in Part 3 of this course, when we build a

custom client and see how it communicates with the server.

MCP versus Function calling

Before MCPs became mainstream (or popular like they are right now), most AI workflows relied on traditional function calling for tools.

Here's a visual that explains Function calling & MCP:



Function calling enables LLMs to execute predefined functions based on user inputs. In this approach, developers define specific functions, and the LLM

determines which function to invoke by analyzing the user's prompt. The process involves:

1. Developers create functions with clear input and output parameters.
2. The LLM interprets the user's input to identify the appropriate function to call.
3. The application executes the identified function, processes the result, and returns the response to the user.

But there are limitations as well:

- As the number of functions grows, managing and integrating them becomes complex. Requires M×N integrations.
- Functions are closely tied to specific applications, making reuse across different systems challenging.
- Any changes require manual updates across all instances where the function is used.

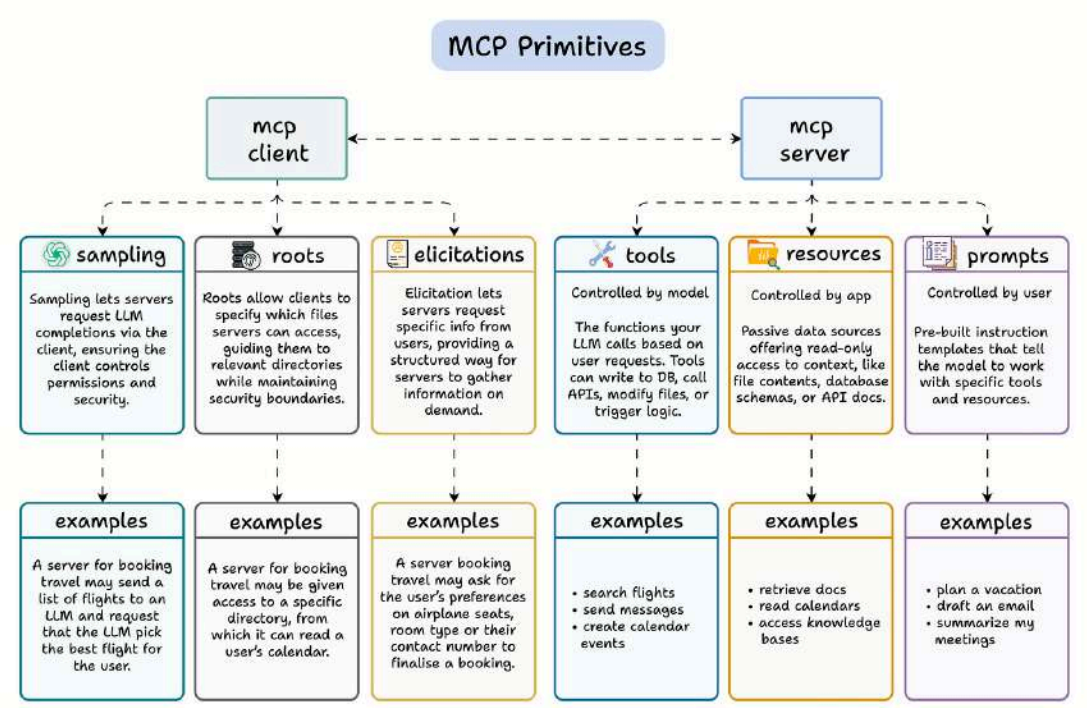
MCP offers a standardized protocol for integrating LLMs with external tools and data sources. It decouples tool implementation from their consumption, allowing for more modular and scalable AI systems.

6 Core MCP Primitives

Developers feel MCP is just another tool calling standard, but that's just scratching the surface.

But unlike simple tool calling, MCP creates a two-way communication between your AI apps and servers.

Here's a breakdown of the 6 core MCP primitives that make MCPs powerful (explained with examples):



Let's start with the client, the entity that facilitates conversation between the LLM app and the server, offering 3 key capabilities:

1) Sampling

The client side always has an LLM.

Thus, if needed, the server can ask the client's LLM to generate some completions, while the client still controls permissions and safety.

For example, an MCP server with travel tools can ask the LLM to pick the optimal flight from a list.

2) Roots

This allows the client to define what files the server can access, making interactions secured, sandboxed, and scoped.

For example, a server for booking travel may be given access to a specific directory, from which it can read a user's calendar.

3) Elicitations

This allows servers to request user input mid-task, in a structured way.

For example, a server booking travel may ask for the user's preferences on airplane seats, room type, or their contact number to finalise a booking.

Moving on, let's talk about the MCP server now.

Server also exposes 3 capabilities: tools, resources, and prompts

4) Tools

Controlled by the model, tools are functions that do things: write to DBs, trigger logic, send emails, etc.

For example:

- search flights
- send messages
- create calendar events

5) Resources

Controlled by the app, resources are the passive, read-only data like files, calendars, knowledge bases, etc.

Examples:

- retrieve docs
- read calendars
- access knowledge bases

6) Prompts

Controlled by the user, prompts are pre-built instruction templates that guide how the LLM uses tools/resources.

Examples:

- plan a vacation
- draft an email

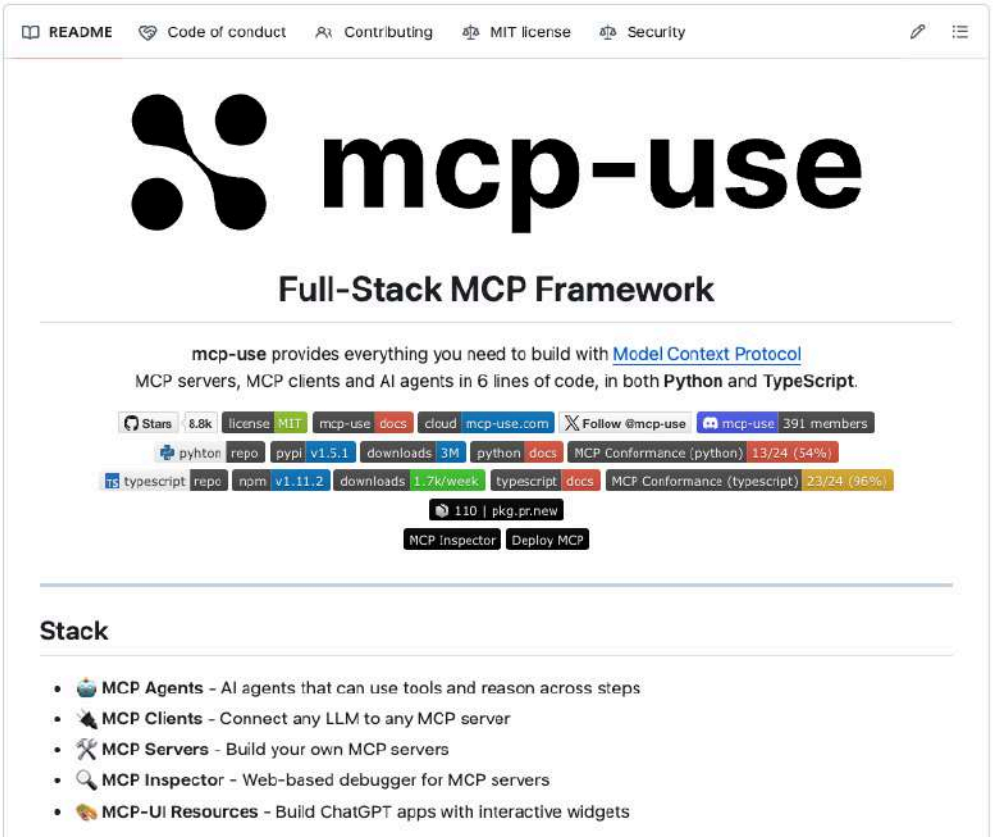
- summarize my meetings

It shows that MCP is not just another tool calling standard. Instead, it creates a two-way communication between your AI apps and servers to build powerful AI workflows.

With the core ideas of MCP in place, we're now ready to see how this translates into real development workflows.

MCP defines the structure, but developers still need a straightforward way to build agents, configure clients and expose capabilities through servers.

This is where the open-source framework mcp-use becomes useful.



It supports the full MCP ecosystem including agents, clients and servers to help build end-to-end workflows suitable for both experimentation and production environments.

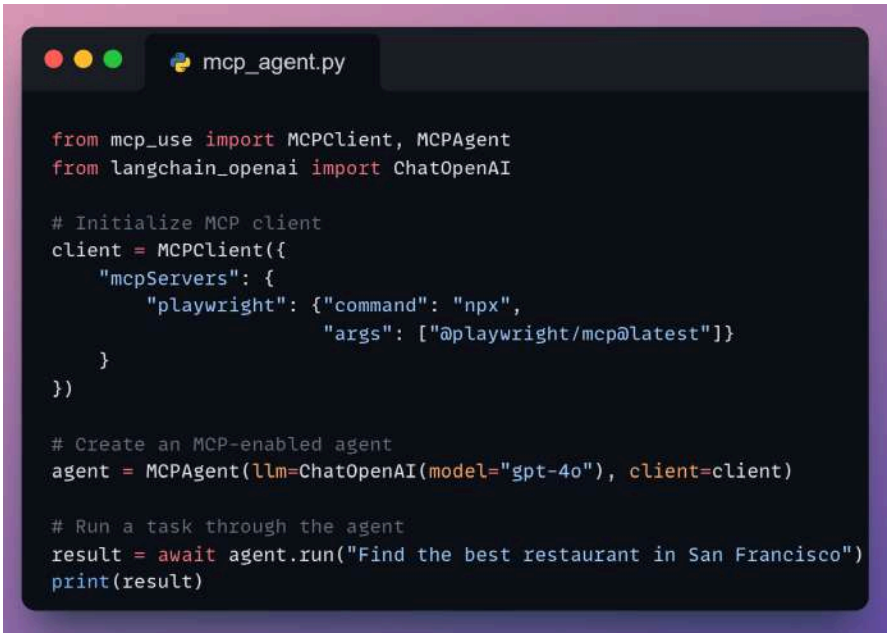
Creating MCP Agents

mcp-use makes it easy to build MCP-enabled agents without handling low-level protocol details yourself.

It sets up the MCP client, connects to one or more servers, discovers available tools, and exposes them to the LLM in a structured way.

This allows the agent to decide when to call a tool, while the framework manages capability loading and communication under the hood.

We can build an mcp-enabled agent using mcp-use in just 6 lines of code:



This creates an MCP client, connects it to a server (Playwright in this example), wraps the server's capabilities as tools, and passes them to an LLM-powered agent.

From here, the LLM can request tool calls naturally during reasoning, while mcp-use handles execution and streaming.

Common Pitfall: Tool Overload

When LLMs gain access to many server tools, certain predictable issues appear:

#1) Tool-name hallucinations

The model may invent a tool that does not exist. This usually happens when the tool list is large or poorly named.

#2) Confusion between similar tools

If a server exposes several tools with overlapping responsibilities, the model may struggle to choose the correct one.

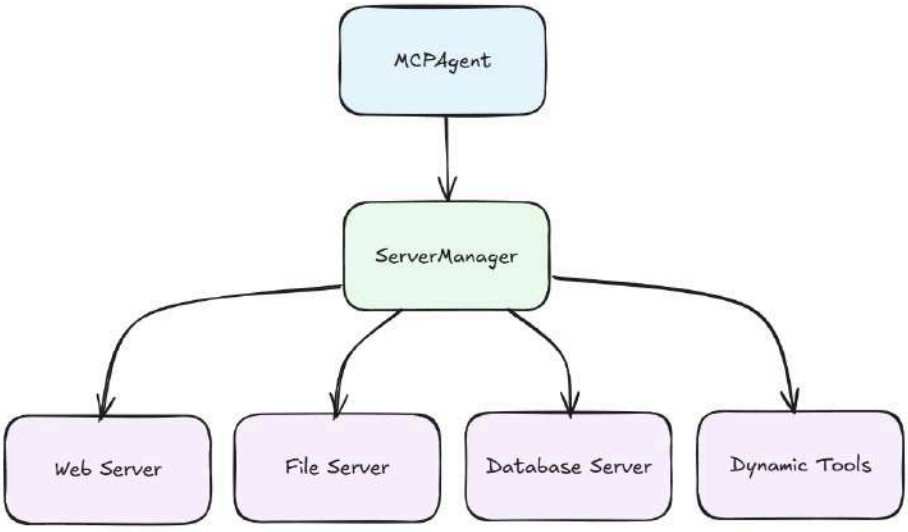
#3) Degraded decision quality with large toolsets

Presenting too many tools at once increases cognitive load for the LLM, leading to inconsistent tool selection or unnecessary calls.

These issues arise from typical LLM behavior when exposed to large toolsets.

Solution: The Server Manager

mcp-use includes a built-in Server Manager, which directly addresses the common failure modes agents face when interacting with large or multi-server toolsets.

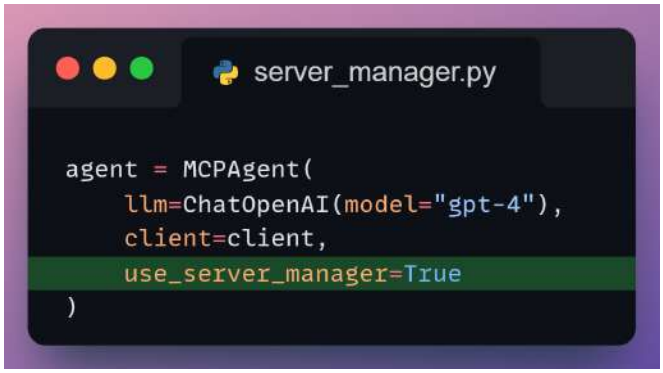


Instead of exposing every tool from every server at once - something that often leads to tool-name hallucinations, confusion between similar tools and degraded reasoning, the Server Manager keeps the agent's active toolset intentionally narrow and context-driven.

When enabled, the Server Manager:

- Loads tools dynamically, only when needed
- Discovers which server is appropriate for the task
- Keeps the active tool list small and focused, reducing model overwhelm
- Updates tools in real time as servers connect or disconnect
- Provides semantic search over all available tools across servers

Enabling it is as simple as setting *use_server_manager=True*.



With this, agents no longer need to juggle dozens of tools at once.

The Server Manager becomes the orchestrator - deciding which server to activate, which tools to load, and when to surface them resulting in clearer tool selection and more stable agent behavior across multi-server environments.

Creating MCP Client

Within mcp-use, every agent embeds an MCP client.

The client is the component responsible for all communication between the agent and any MCP server.

It manages the transport layer and ensures that capabilities (tools, resources, prompts, etc.) remain synchronized throughout an interaction.

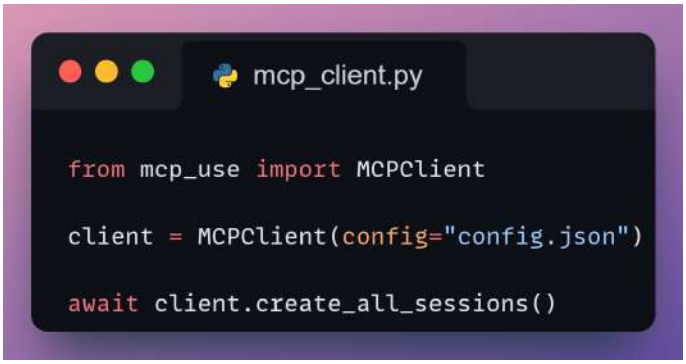
At a high level, the MCP client handles:

- Connecting to MCP servers
- Performing the initial capability handshake
- Streaming tool calls and tool responses
- Retrieving resources
- Receiving notifications
- Routing elicitation requests back to the user or host application

In short, the client keeps the agent and server aligned, ensuring both sides speak the same protocol.

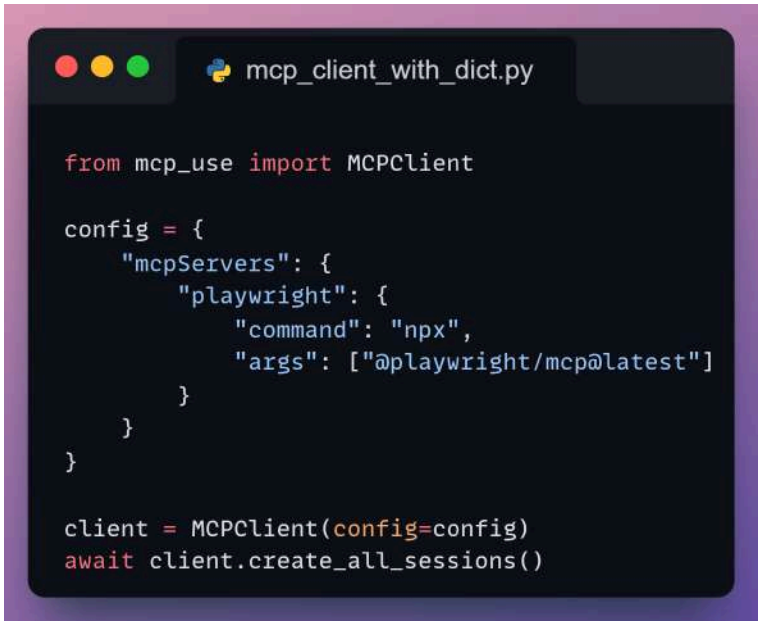
mcp-use provides simple ways to create an MCP client depending on how you prefer to manage server settings.

#1) Load Configuration From a File



This approach is ideal when working with multiple environments or when you prefer keeping server settings version-controlled.

#2) Create From a Python Dictionary



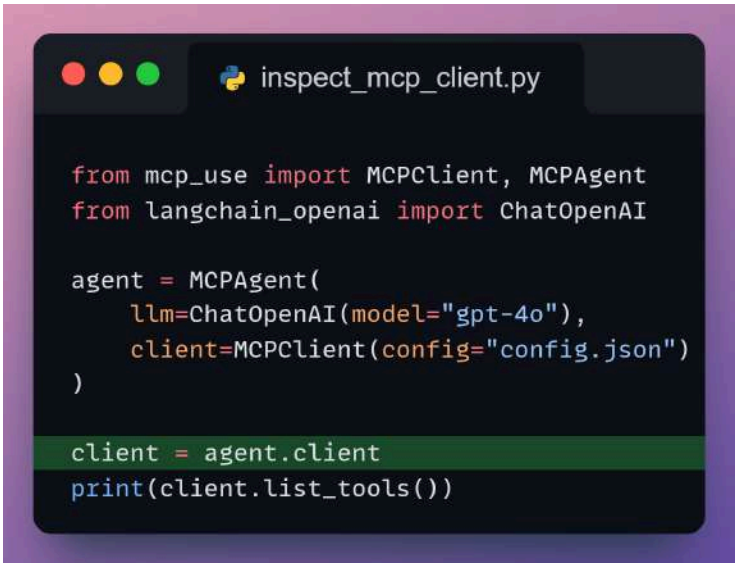
This mirrors the same structure as configuration files but allows programmatic customization inside Python.

Inspecting the Client

Although the agent manages the client internally, mcp-use still allows you to inspect the client directly when needed.

For example, to debug capability discovery or understand which tools are

available, we can inspect using this:



MCP Server

Agents and clients decide how to act, but MCP servers are what make those actions possible.

A server is the source of truth for capabilities - tools to execute, resources to read, prompts to supply, and structured interactions like sampling or elicitation.

Once a server exposes these capabilities, any standard MCP client can discover and use them.

mcp-use provides a lightweight framework for defining these capabilities declaratively, making it easier to build servers that integrate cleanly with MCP agents.

Note: All examples in this section use TypeScript.

The following sections walk through how to create, run, test and deploy MCP servers using mcp-use.

1) Project Generator: create-mcp-use-app

mcp-use includes a project generator that lets you create a new server project with these commands:

Running this creates a ready-to-use server with:

- A TypeScript entrypoint
- Example tools, prompts and resources
- Configuration files
- Built-in support for the MCP Inspector

This provides a convenient starting point for building an MCP server.

2) Exposing MCP Capabilities

Earlier we discussed the six core MCP primitives that power the protocol.

Using mcp-use, an MCP server can expose them - tools, resources, prompts, sampling, elicitation and notifications through small, declarative definitions.

This keeps the server's surface area simple to describe and easy for agents to discover automatically during capability negotiation.

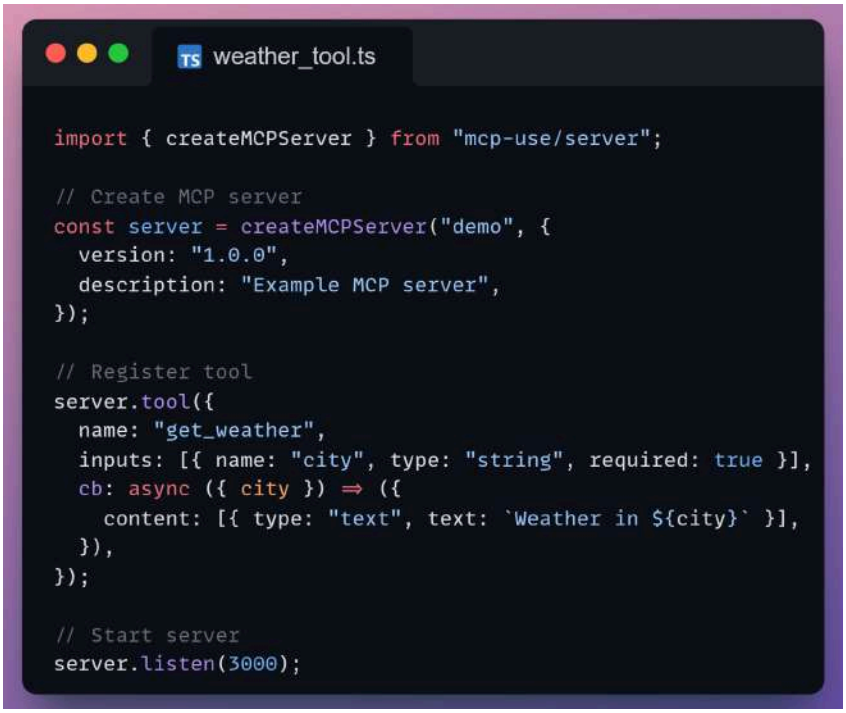
Below are simple examples to understand each capability.

Tools

Tools represent actions the agent can perform. They are the main way an MCP server exposes functionality - anything from API calls to calculations to workflow steps.

In mcp-use, tools are registered on the server using a simple declarative definition.

Each tool includes a name, input parameters and a callback that returns content to the client. Here is a minimal tool example:



This example defines a complete MCP server with a single tool: get_weather, which returns a basic weather response.

Any MCP-compatible client can automatically discover and invoke this tool during capability negotiation.

Resources

Resources expose read-only content such as files or generated text through a stable URI.

Clients can fetch this content at any time during a session.

Prompts

Prompts define reusable instruction templates that agents can invoke to generate structured messages.

They let your server provide consistent, well-formed prompts for common tasks.

Sampling

Sampling lets your server ask the client's model to generate text mid-workflow.

It's useful when the server needs the model to decide, summarize or choose between options.

Elicitation

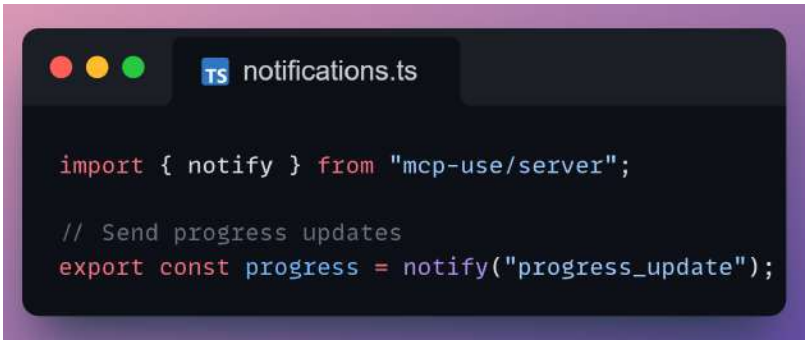
Elicitation requests structured input from the user, such as selecting an option or entering text.

This enables interactive workflows where the server needs clarification or a choice.

Notifications

Notifications allow your server to push asynchronous updates such as progress or status changes to the client.

They're ideal for long-running or multi-step operations.

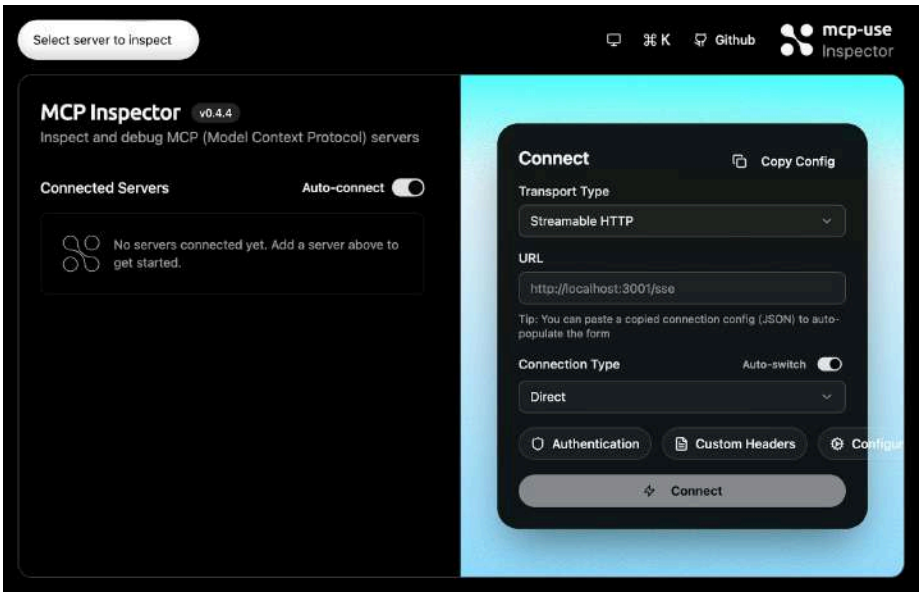


Together, these primitives cover the full MCP surface: operations, structured context retrieval, user interactions and asynchronous messaging.

Any MCP client discovers these automatically during capability negotiation which means your server becomes instantly usable by agents without extra configuration.

3) MCP Inspector

When you start your server in development mode (*npm run dev*), mcp-use automatically launches the MCP Inspector, a web-based dashboard for inspecting and debugging MCP servers.



The Inspector lets you:

- Browse and test tools interactively
- Explore resources and inspect their content
- Preview prompts and validate arguments
- Watch sampling and notification events in real time
- Monitor all JSON-RPC traffic between client and server

It's the fastest way to verify your server's capabilities before connecting it to an agent.

4) MCP-UI

MCP-UI is a UI framework for MCP servers, enabling them to expose simple UI widgets that appear inside compatible clients.

These widgets let your server surface status, previews or quick visual outputs without requiring a full application.

In mcp-use, you define these widgets through a small, focused API.

Here's a simple example:

Widgets are useful for things like:

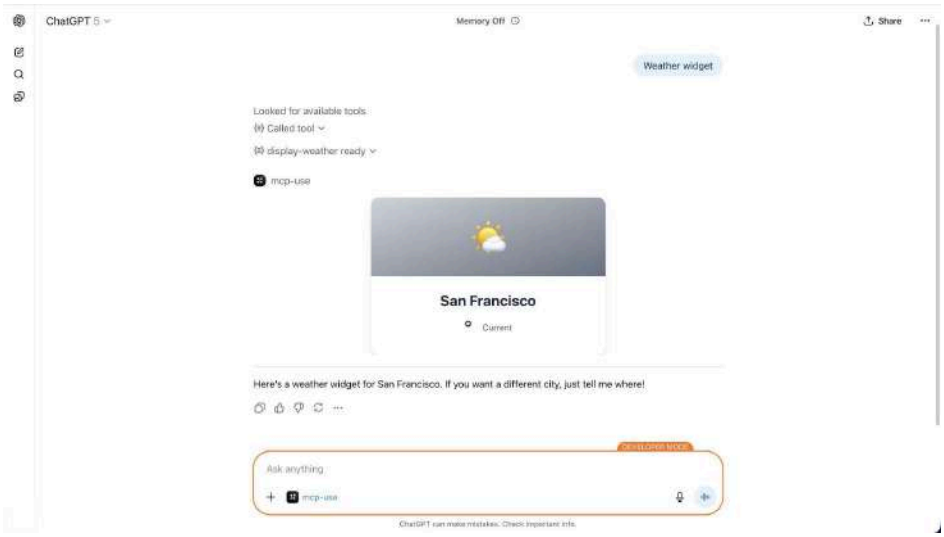
- Server health indicators
- Resource previews
- Results from recent tool calls
- Debugging or introspection output

They're optional, but they significantly enhance the developer experience when building or testing MCP servers.

5) Apps SDK

The Apps SDK is OpenAI's framework for building interactive UI widgets that appear directly inside ChatGPT or other Apps-SDK-compatible clients.

These widgets are written in React and allow tools to return interfaces such as cards, previews or small apps rather than plain text.



MCP servers can expose these widgets as capabilities, enabling richer workflows with minimal overhead.

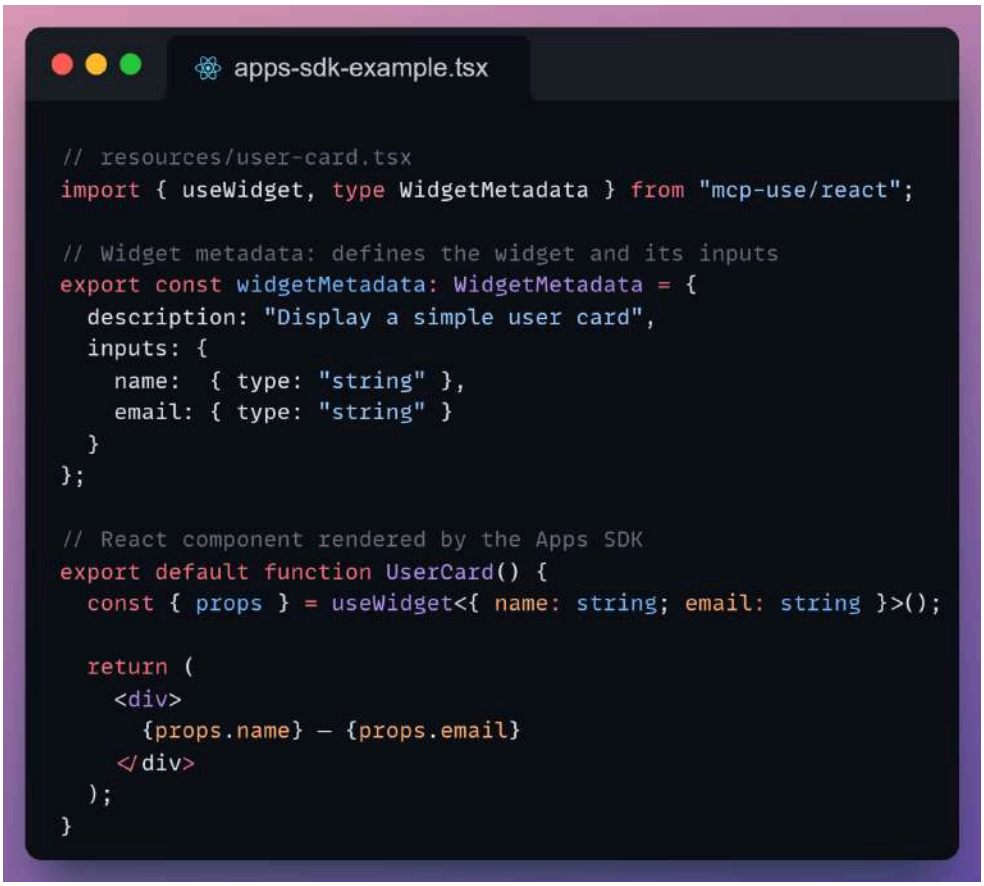
mcp-use simplifies this process.

Instead of manually registering widgets, writing HTML templates, configuring CSP, and bundling assets, you can place a React component in the *resources/* directory with a *widgetMetadata* export. mcp-use will:

- Scan the folder at server startup
- Extract the component metadata
- Register the widget as a resource (and a tool if the widget defines inputs)
- Bundle it for the Apps SDK

- Apply the required CSP configuration
- Provide the useWidget hook for accessing props, output, theme and state

Below is a minimal example of an Apps SDK widget.



It defines metadata (so the server can expose the widget as a capability) and a React component (which the client renders inside ChatGPT).

6) Tunneling

During development, MCP servers usually run on a local machine. When an external MCP client such as ChatGPT, Claude or a mobile agent needs to connect, a public URL is required.