2025 EDITION

FREE

AI Engineering

System Design Patterns for LLMs, RAG and Agents

Daily Dose of Data Science

Akshay Pachaar & Avi Chawla

DailyDoseofDS.com

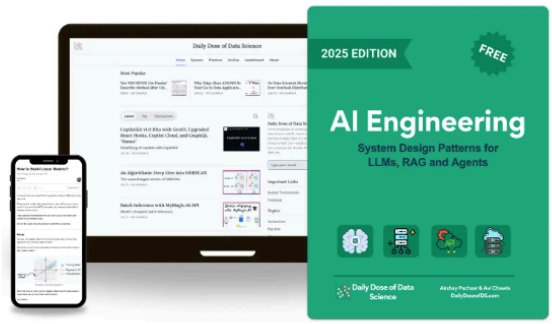
How to make the most out of this book and your time?

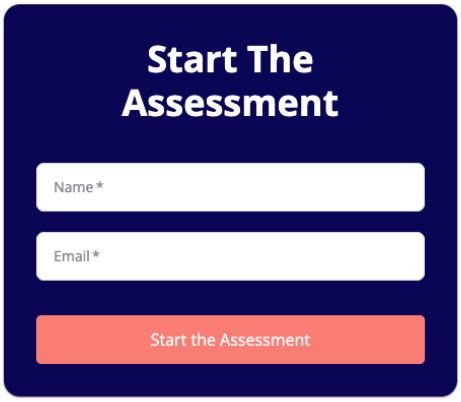
The reading time of this book is about 20 hours. But not all chapters will be of relevance to you. This 2-minute assessment will test your current expertise and recommend chapters that will be most useful to you.

Are you prepared for a career in AI Engineering?

Answer 15 yes/no questions, and we'll email you the list of chapters that you must read to improve your AI Engineering skillset.

Start the Assessment Below!





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."

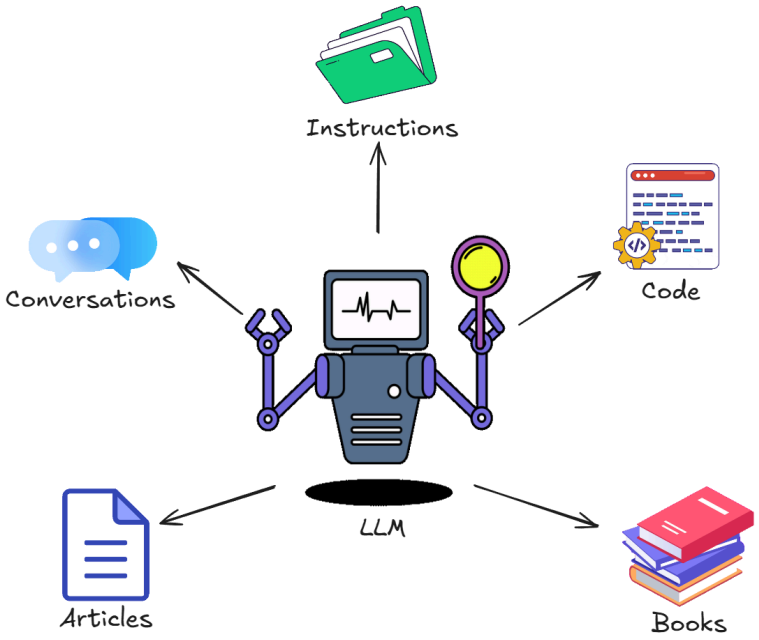
Or they say:

"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.

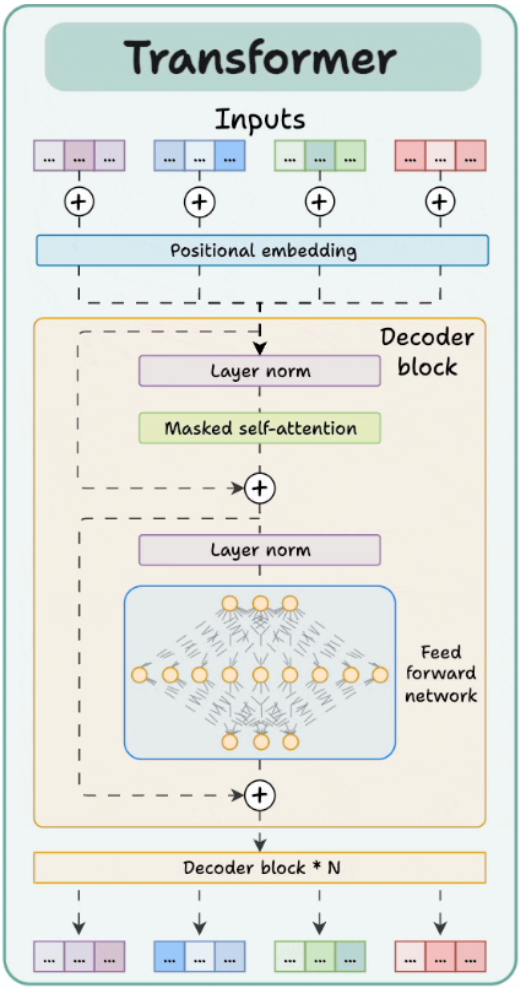


With enough exposure, the model becomes remarkably good at continuing any piece of text in a coherent, meaningful way.

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.

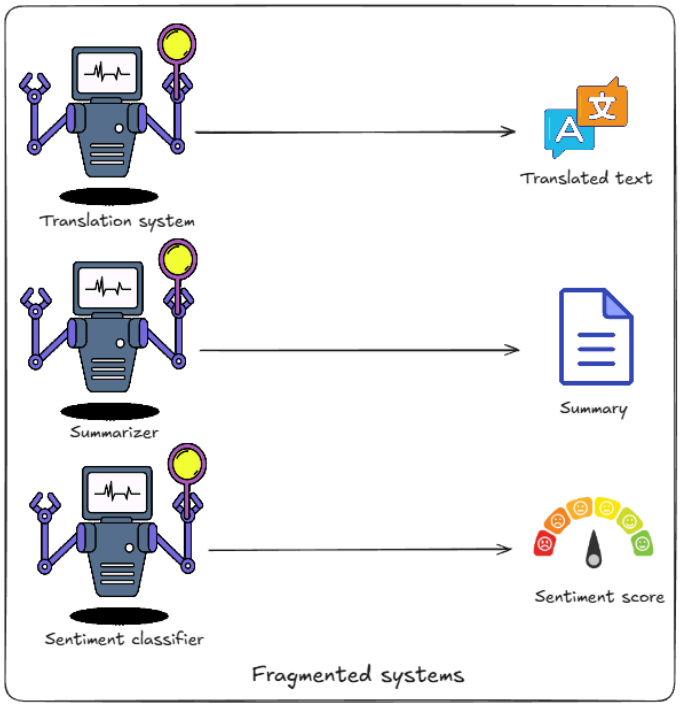
Everything an LLM does from summarizing a document, generating a function or explaining a concept emerges from choosing the next token that best fits the patterns it has learned.



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.

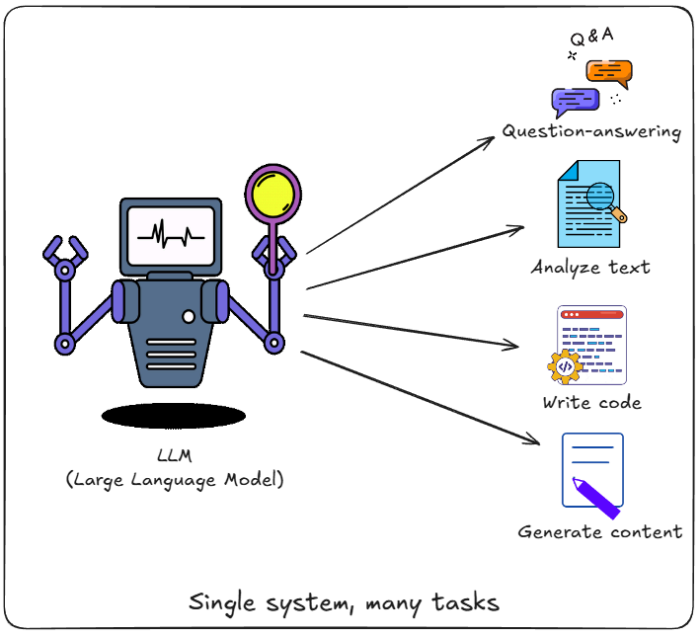


- A translation system handled only translation.
- A summarizer knew only summarization.
- A sentiment classifier recognized only sentiment.

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.

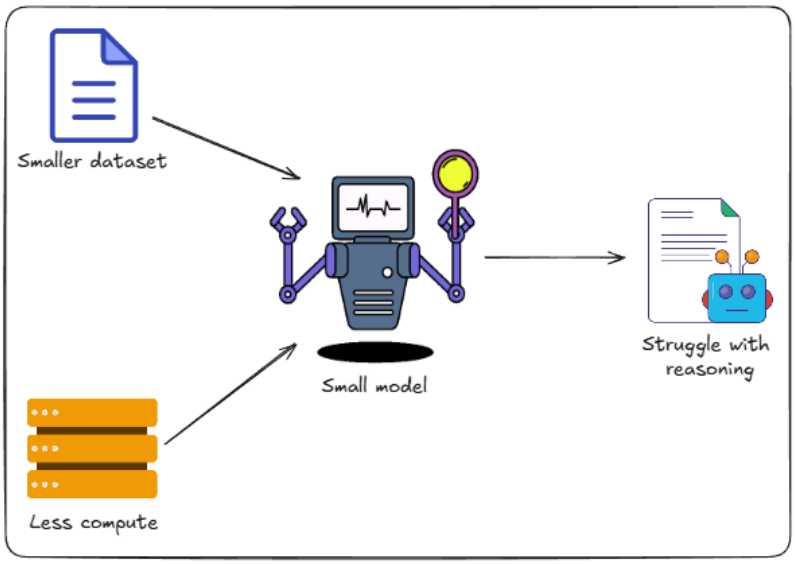
What makes an LLM 'large' ?

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

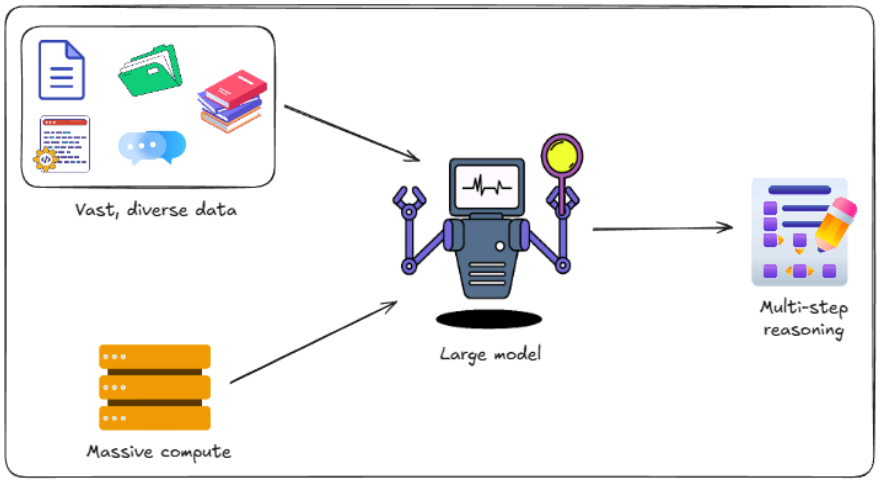
Parameters are the internal values that the model adjusts during training. Each parameter represents a small piece of the patterns the model has learned.

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.

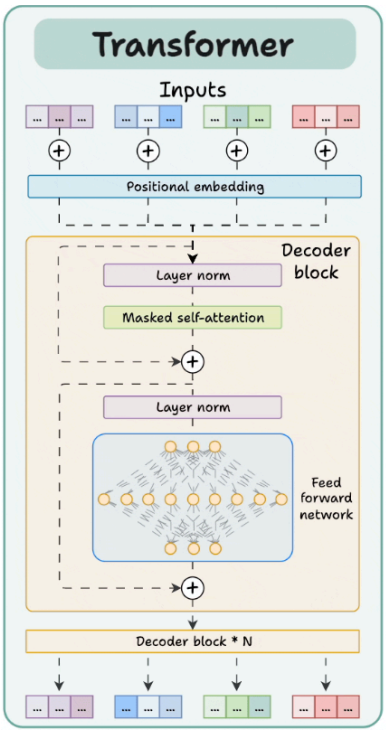
This effect held consistently: models with more parameters, trained on broader data, produced more reliable, coherent and adaptable outputs.

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

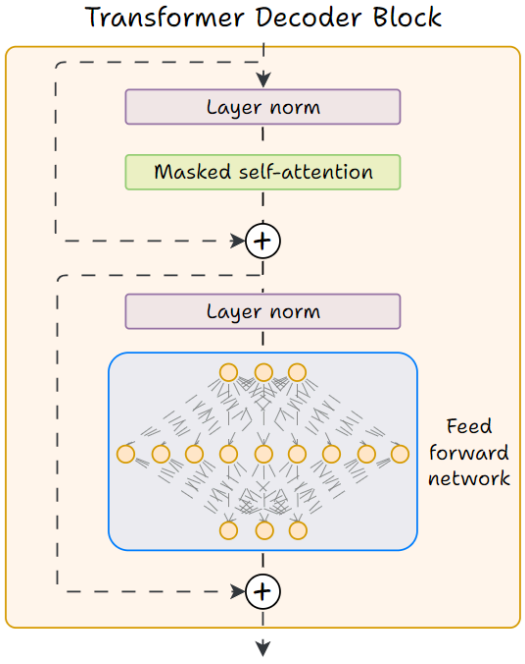
Text is first broken into tokens. A token may be a word or part of a word, depending on how common it is.

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.

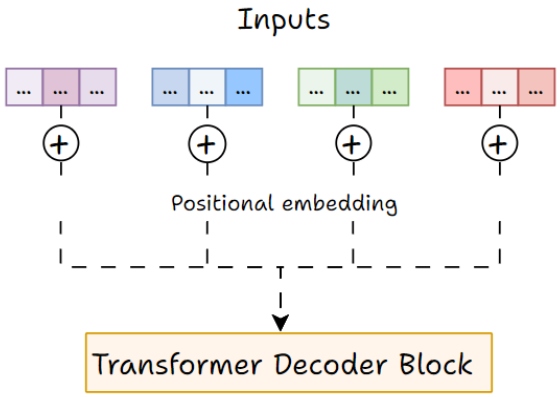


Each layer refines the model's understanding by comparing tokens, attending to important parts of the input, and updating their representations.

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.

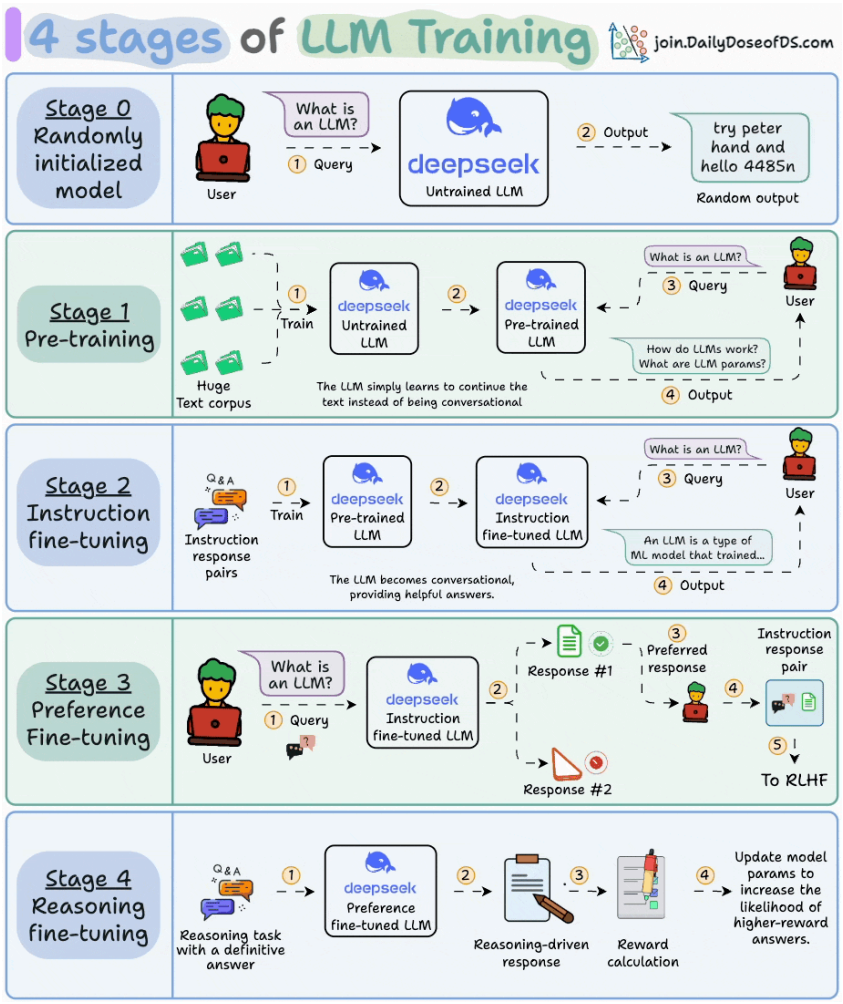
How to train LLM from scratch?

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.

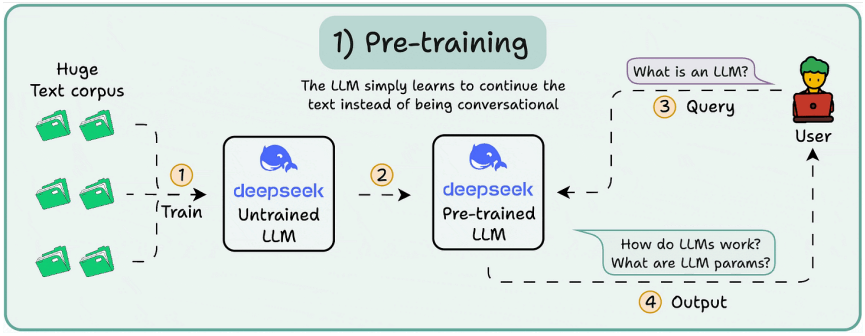
You ask, "What is an LLM?" and get gibberish like "try peter hand and hello 448Sn".

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

1) Pre-training

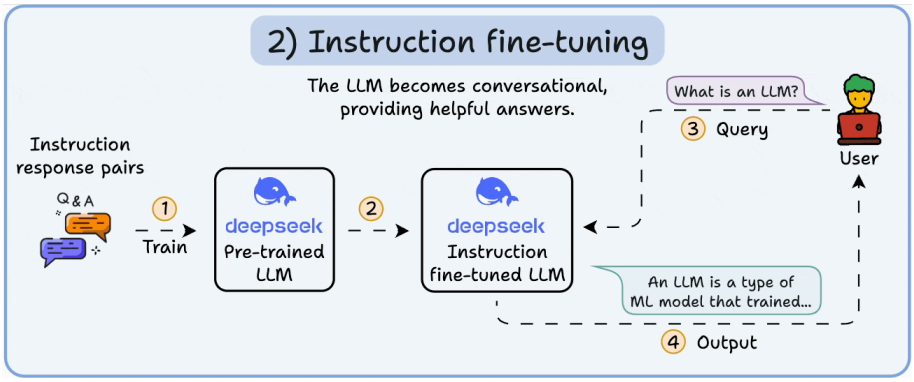
This stage teaches the LLM the basics of language by training it on massive corpora to predict the next token. This way, it absorbs grammar, world facts, etc.

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:

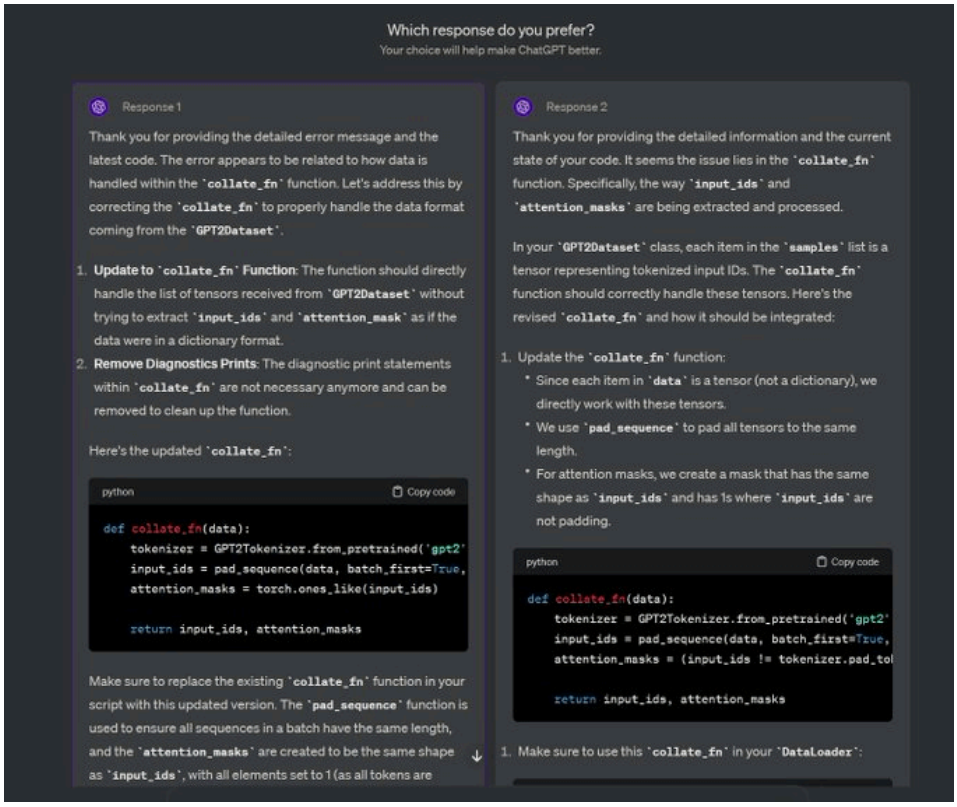
- Utilized the entire raw internet archive and knowledge.
- The budget for human-labeled instruction response data.

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?



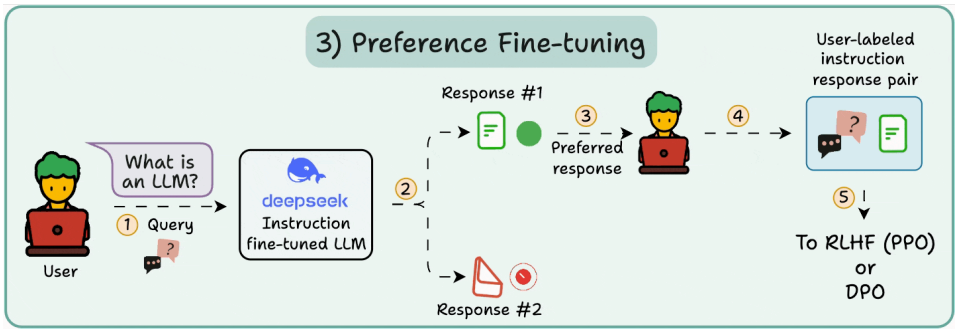
That's not just for feedback, but it's valuable human preference data.

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.

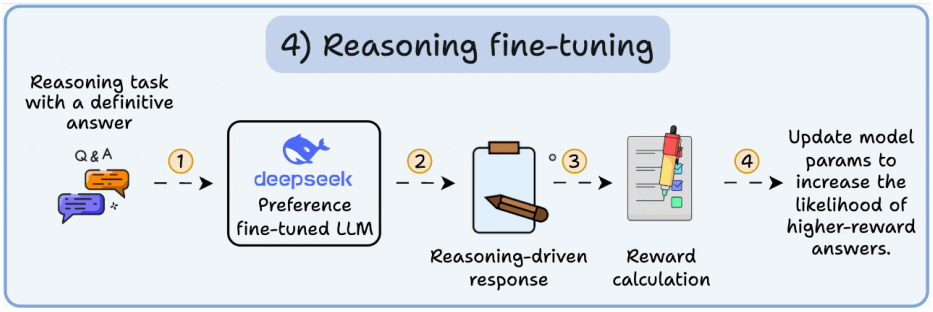
But we can improve the LLM even more.

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.

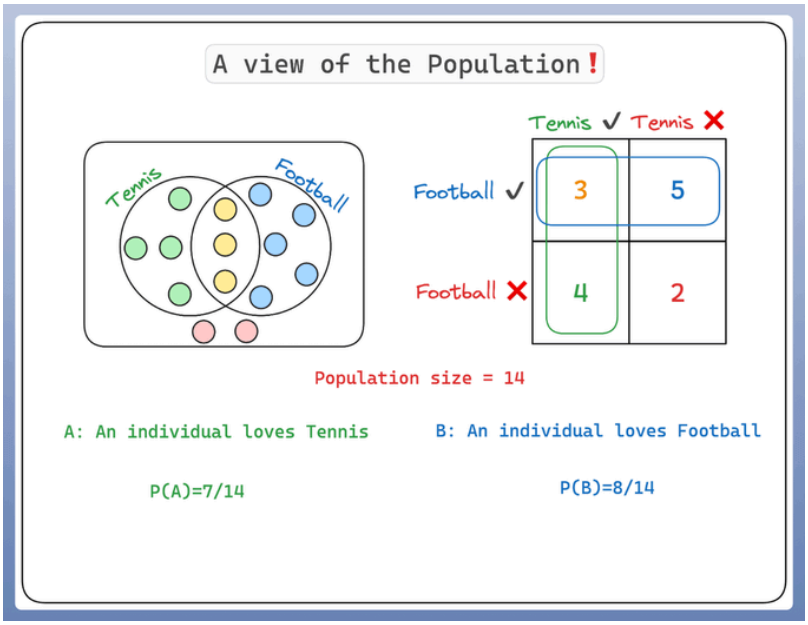
- Start with a randomly initialized model.
- Pre-train it on large-scale corpora.
- Use instruction fine-tuning to make it follow commands.
- Use preference & reasoning fine-tuning to sharpen responses.

How do LLMs work?

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

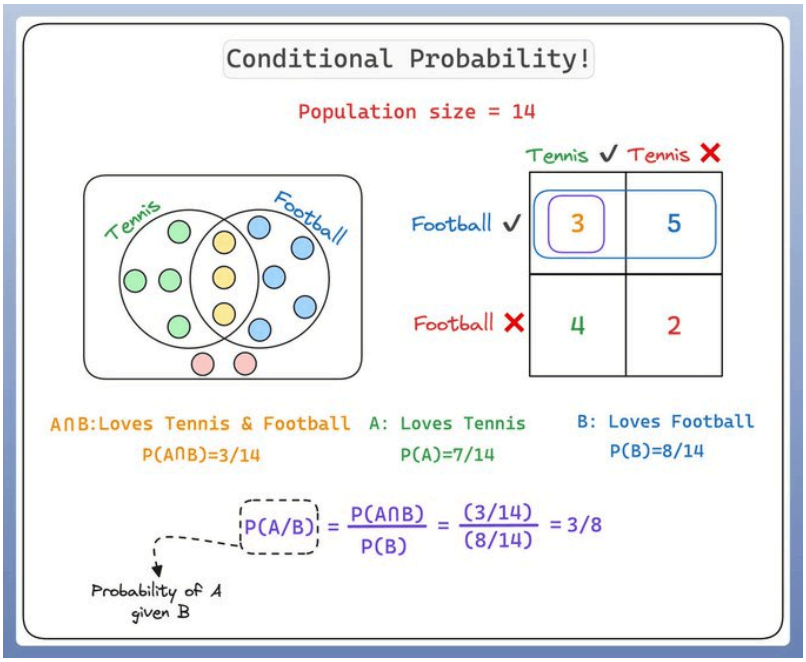
Before diving into LLMs, we must understand conditional probability.

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"

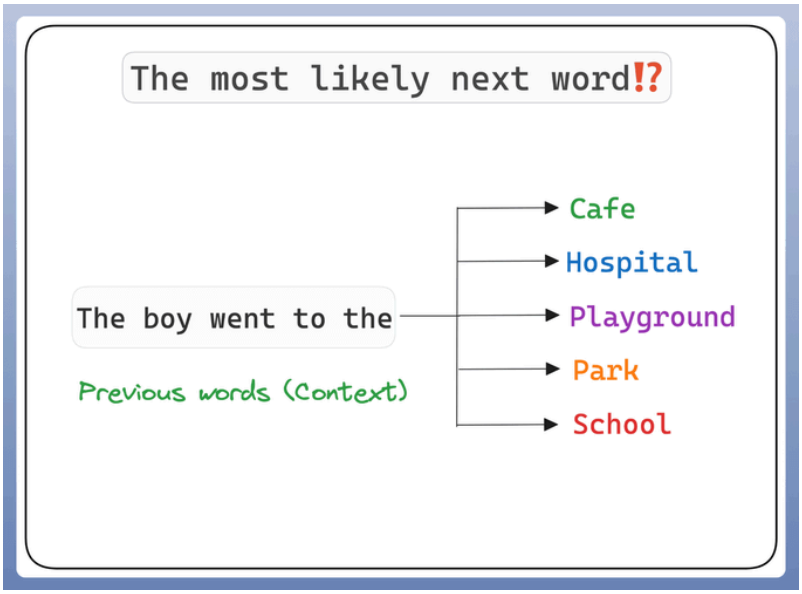
For instance, if we're predicting whether it will rain today (event A), knowing that it's cloudy (event B) might impact our prediction.

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!

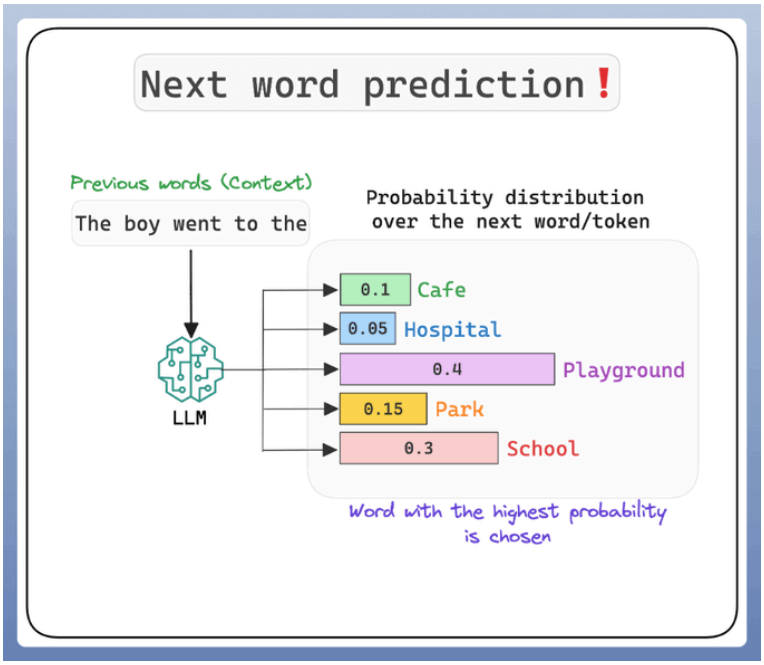
Now, how does this apply to LLMs like GPT-4?

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



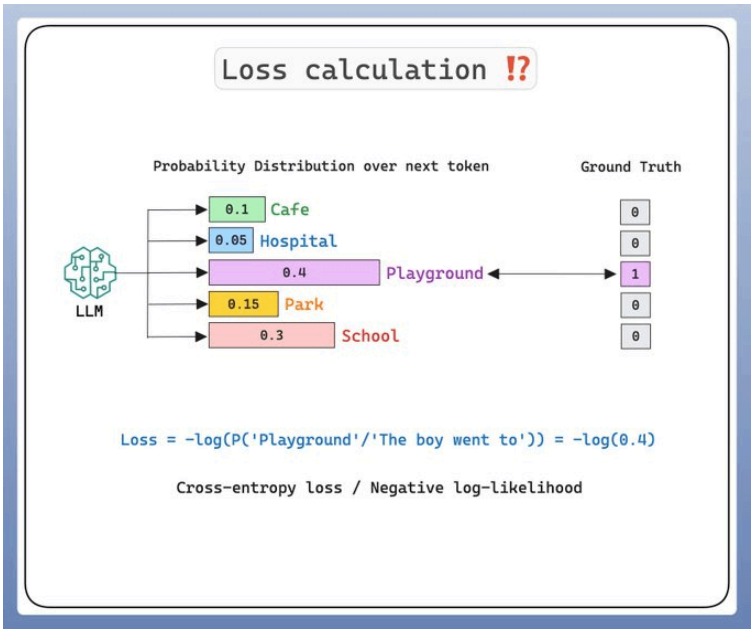
This is a question of conditional probability: given the words that have come before, what is the most likely next word?

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.

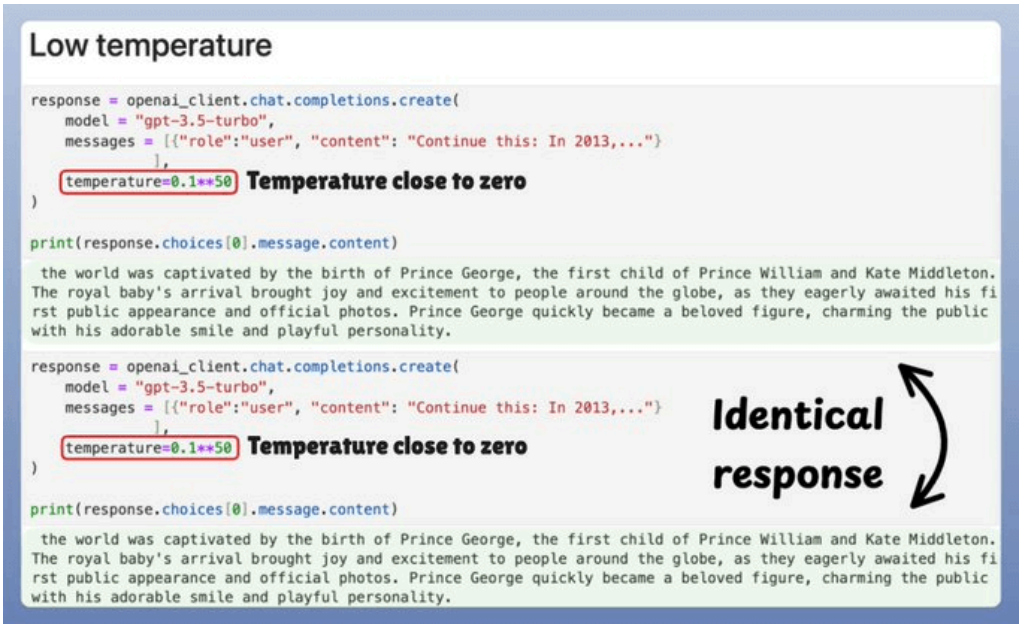


And the parameters of this distribution are the trained weights!

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

But there is a problem!

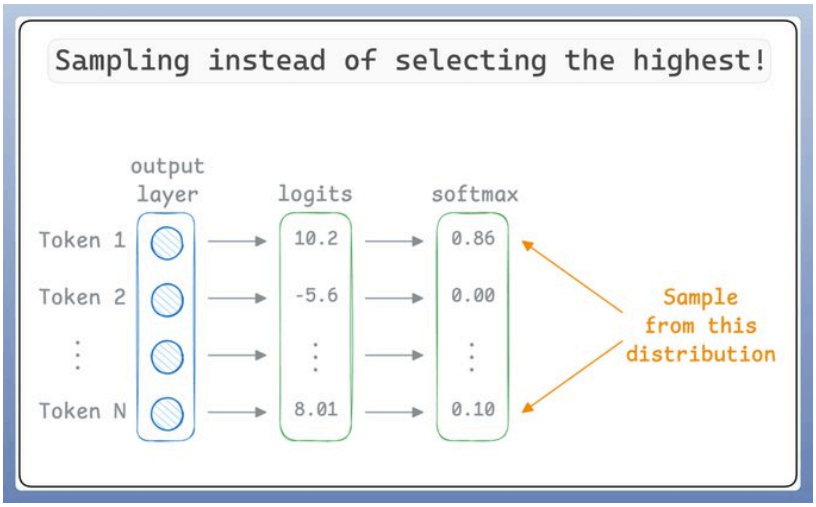
If we always pick the word with the highest probability, we end up with repetitive outputs, making LLMs almost useless and stifling their creativity.



This is where temperature comes into the picture.

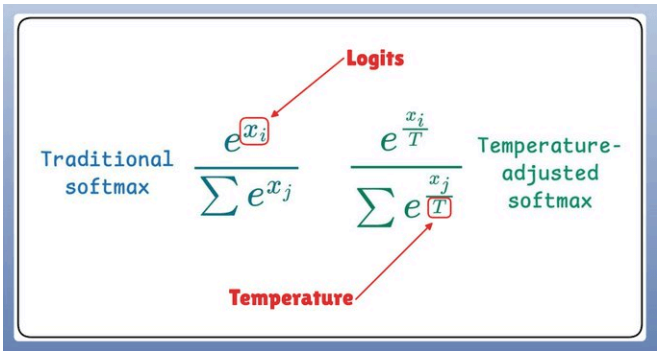
Let's understand what's going on..

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

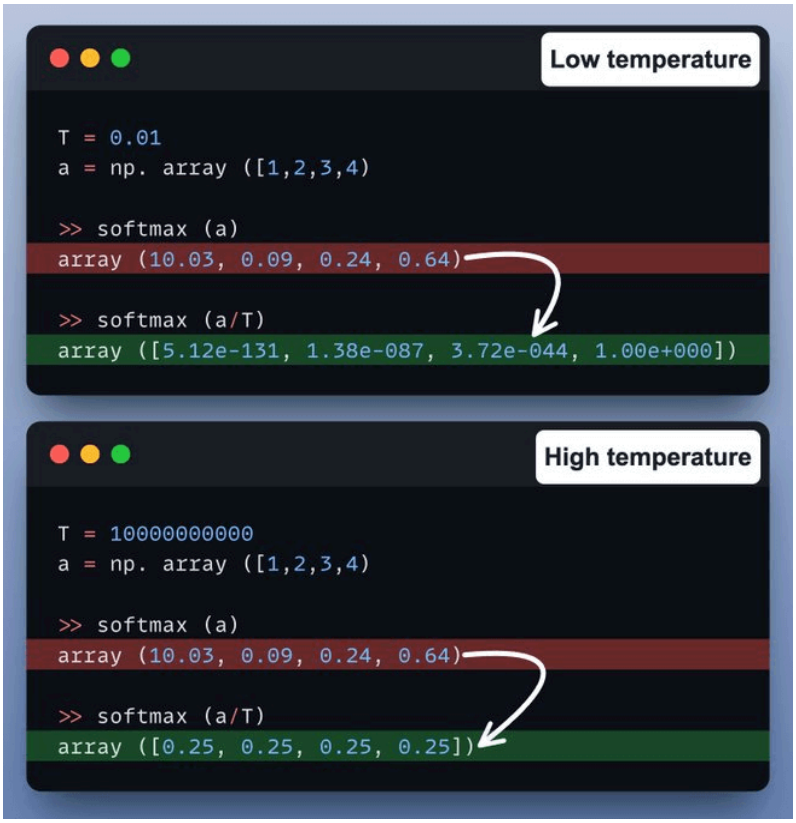


So even if “Token 1” has the highest score, it may not be chosen since we are sampling.

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.

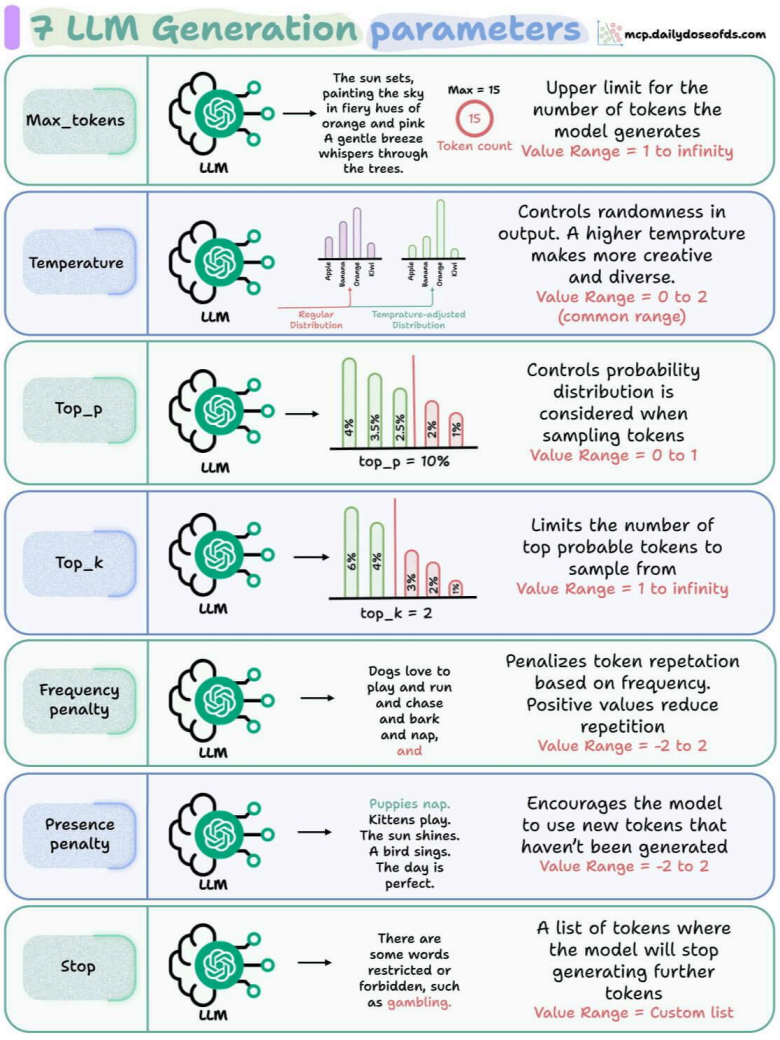
And that is how LLMs work and generate text.

7 LLM Generation Parameters

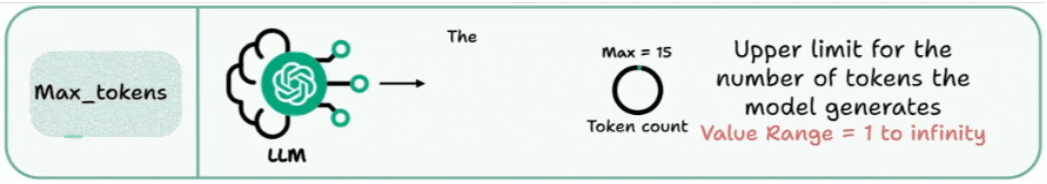
Every generation from an LLM is shaped by parameters under the hood.

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

Here are the 7 levers that matter most:



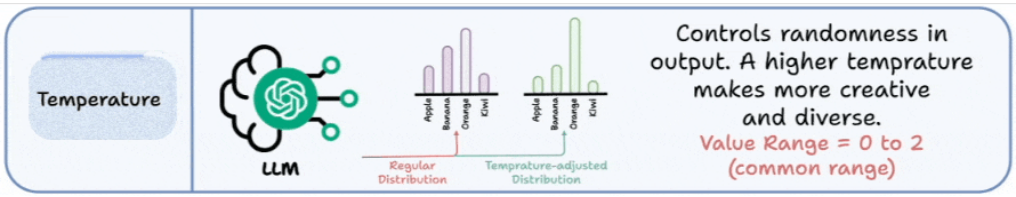
1) Max tokens



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

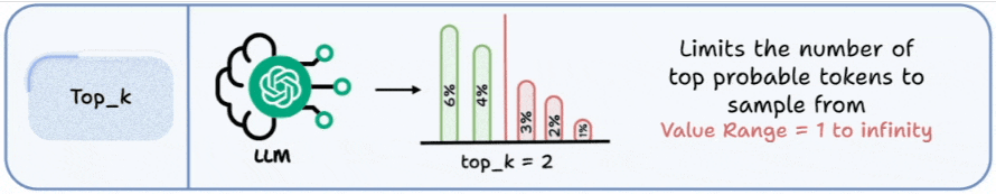


Governs randomness. Low temperature (~0) makes the model deterministic.

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



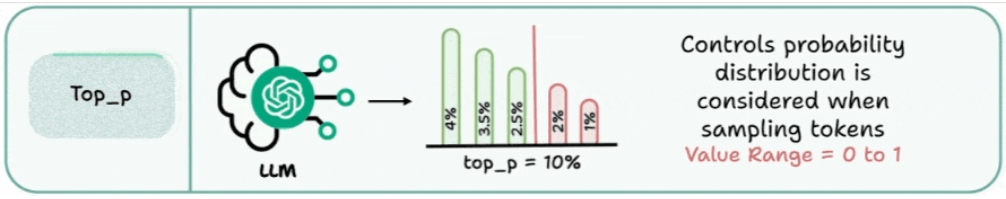
The default way to generate the next token is to sample from all tokens, proportional to their probability.

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)

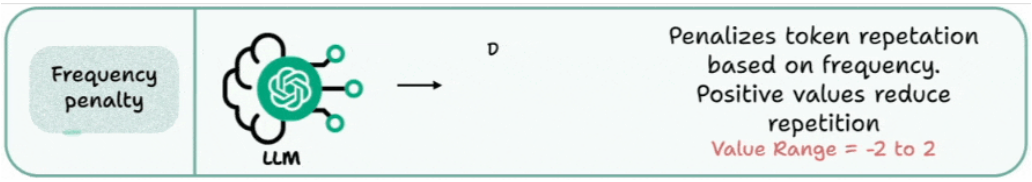


Instead of picking from all tokens or top k tokens, model samples from a probability mass up to p.

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.

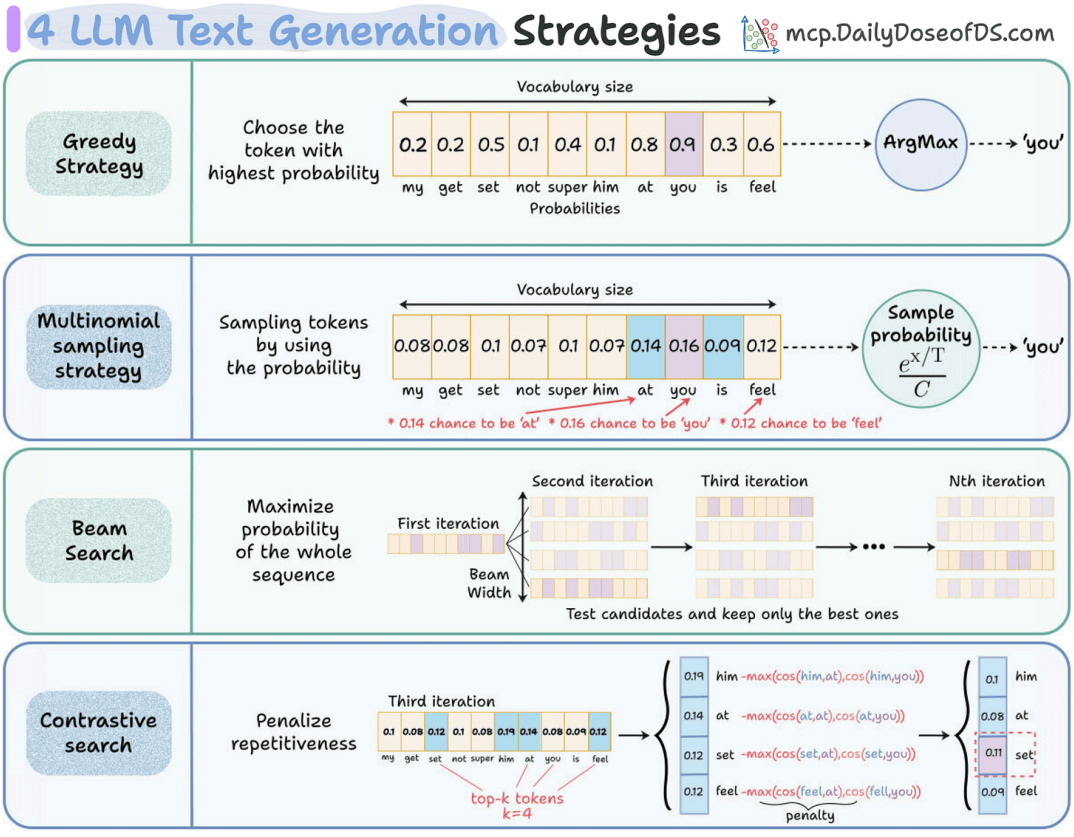
4 LLM Text Generation Strategies

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:

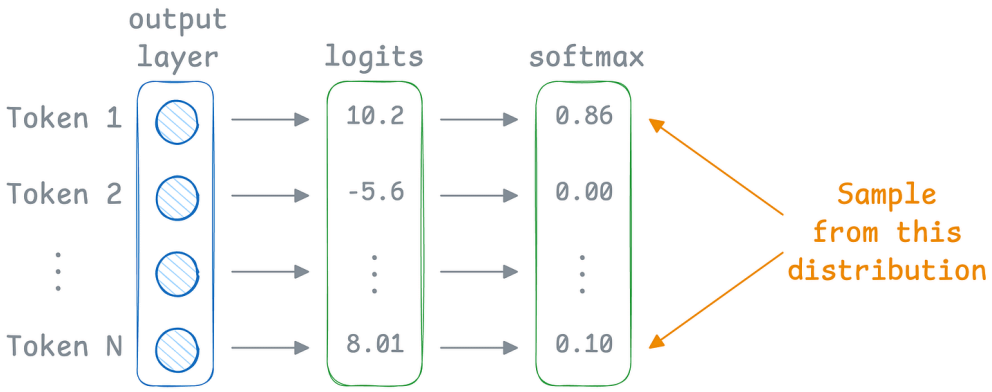


Approach 1: Greedy strategy

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

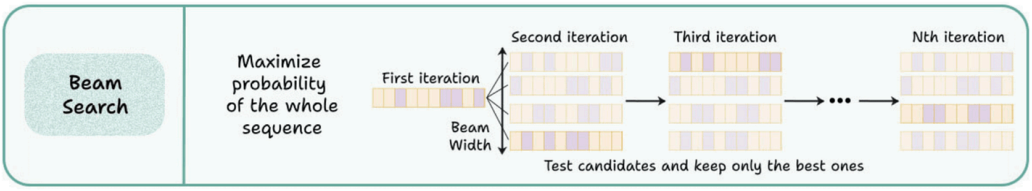
Both approach 1 and approach 2 have a problem. They only focus on the most immediate token to be generated. Ideally, we care about maximizing the probability of the whole sequence, not just the next token.

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

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:



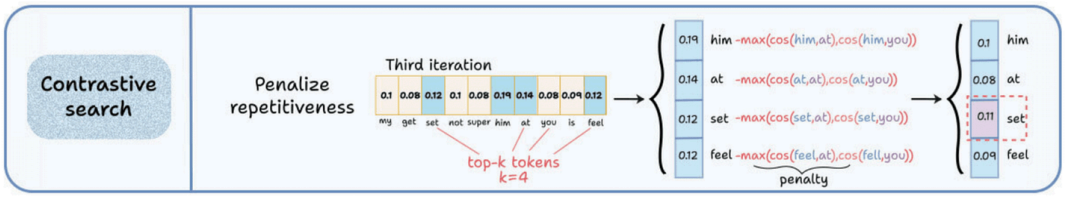
At each step, it expands the top k partial sequences (the beam).

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.

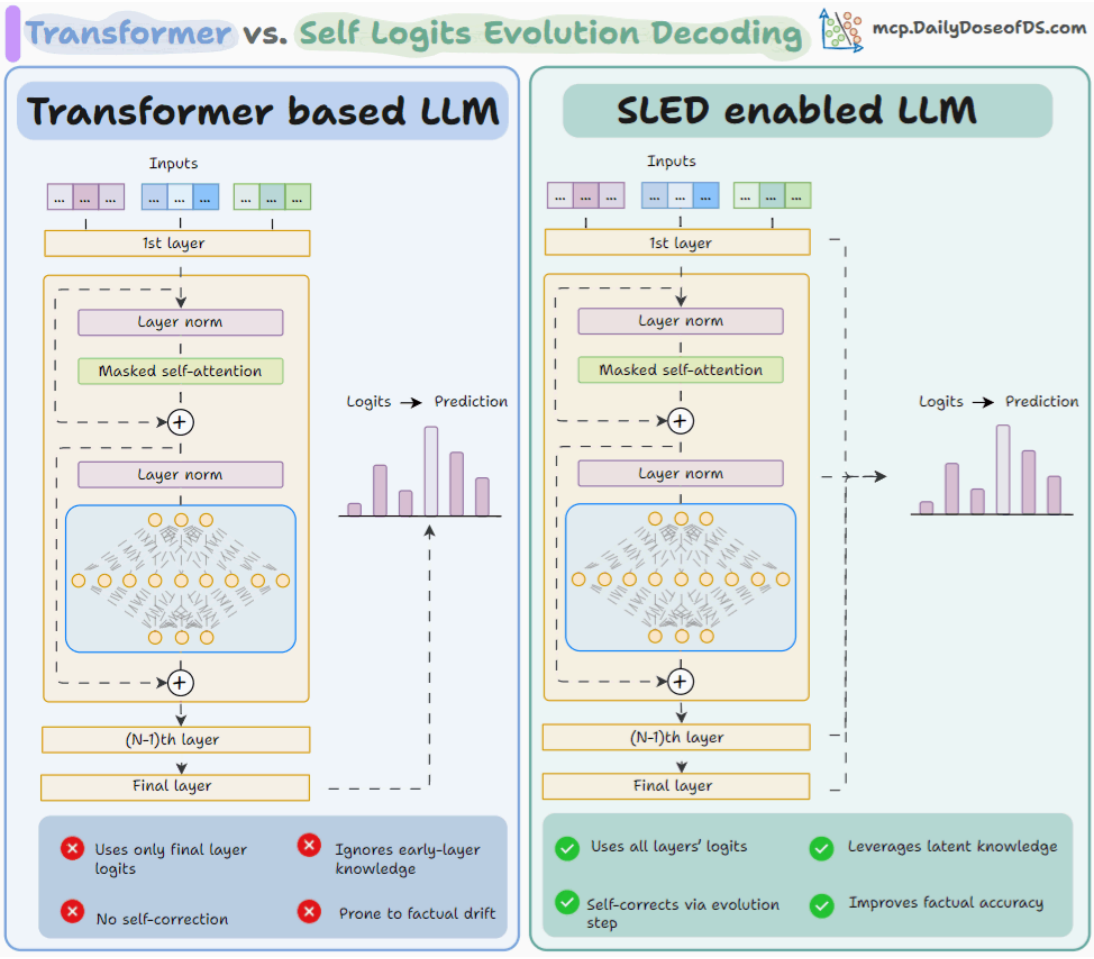
Selects the token that balances probability and diversity.

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.

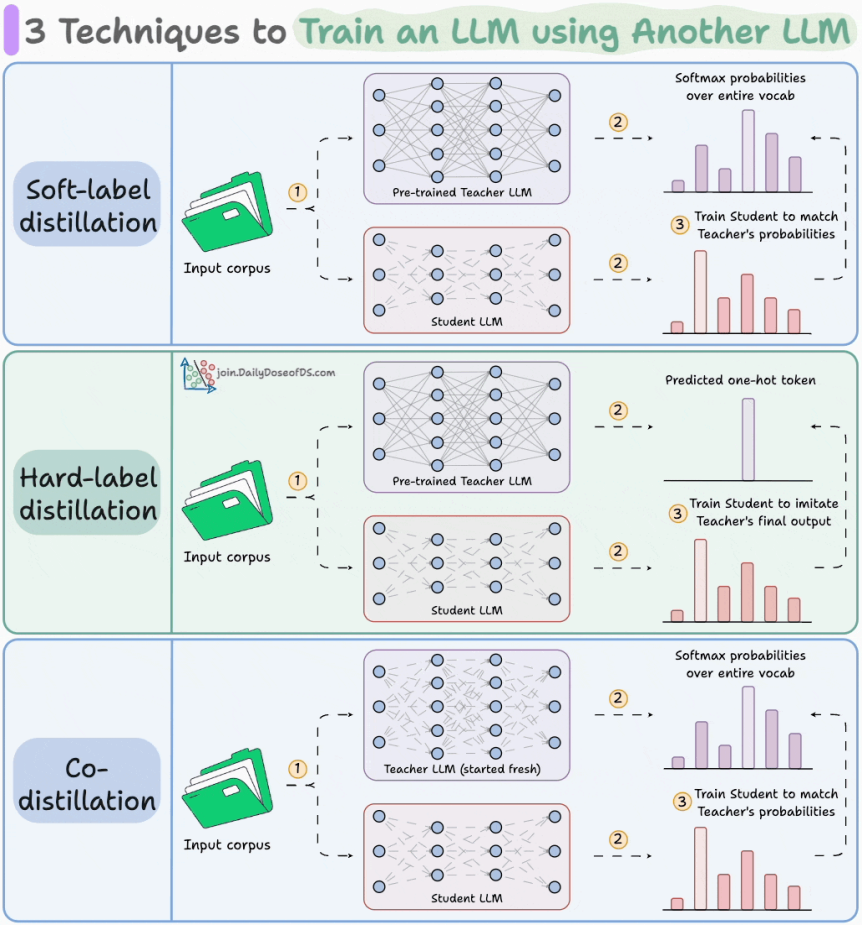
This requires no retraining, extra data or additional compute. By leveraging the model's full internal knowledge rather than only its last step, SLED produces more grounded and factual generations with the same architecture.

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

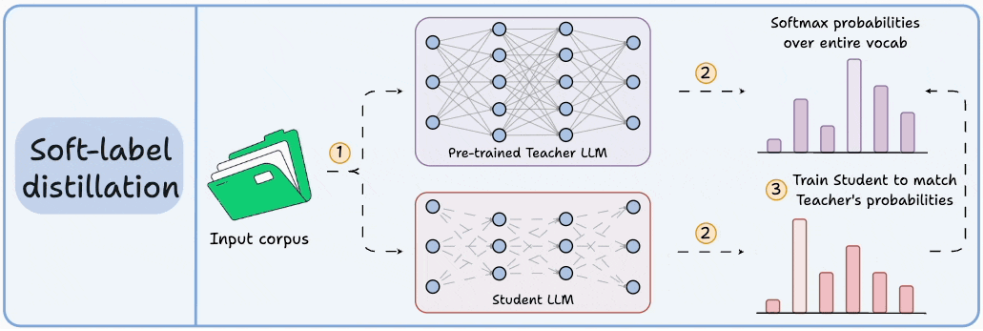
Distillation in LLMs can happen at two stages:

- 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.

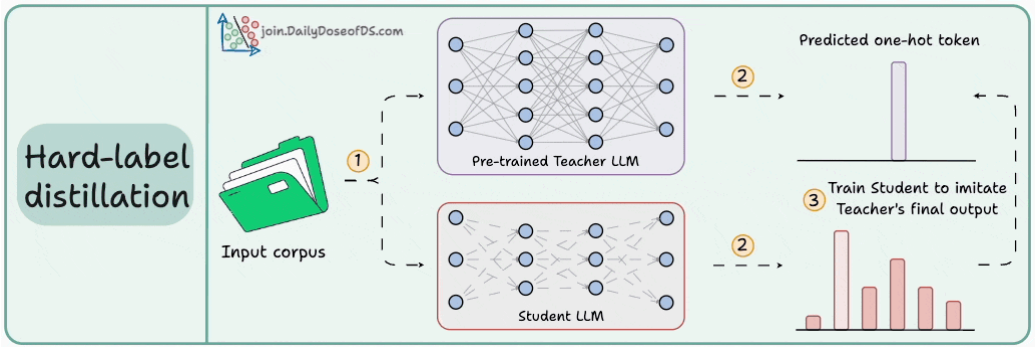
Even if you have access, there's another problem!

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.

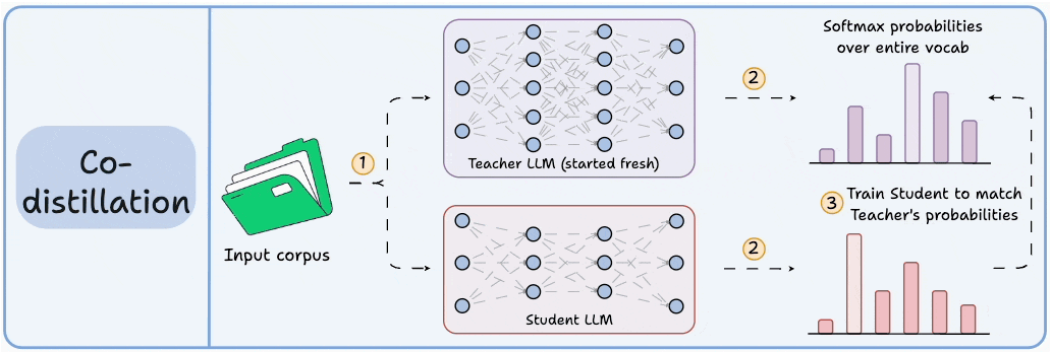
2) Hard-label distillation



- 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

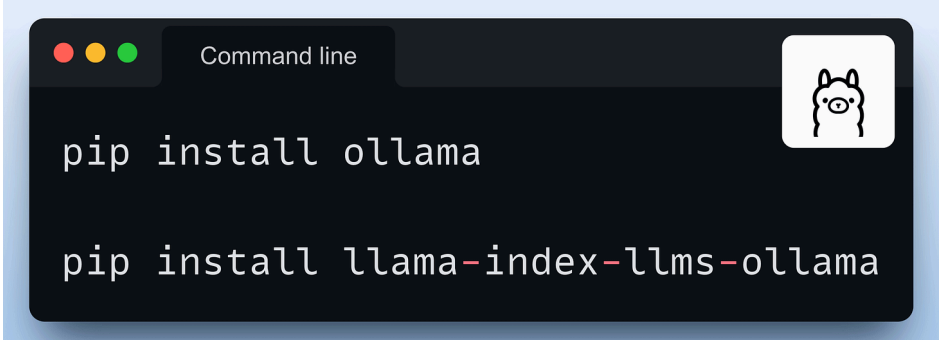
Running a model through Ollama is as simple as executing this command:

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*)

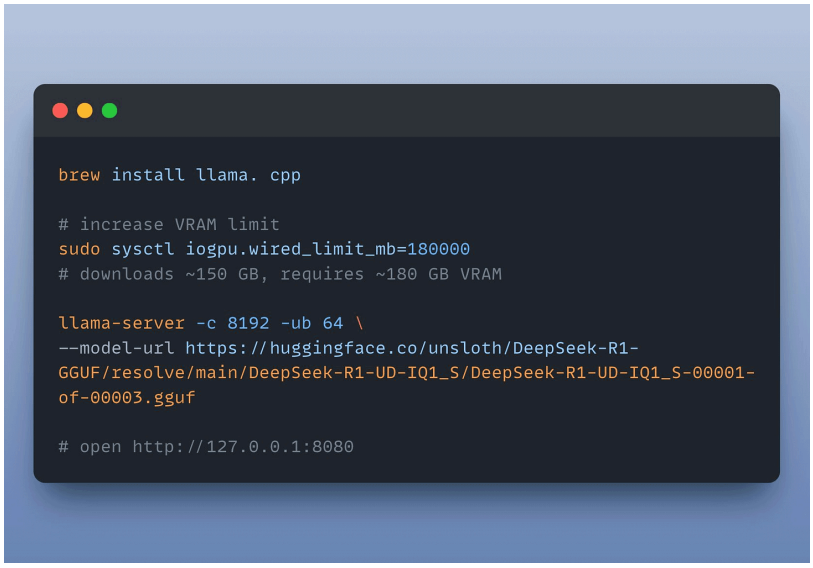
With just a few lines of code, you can locally run LLMs (like DeepSeek) in an OpenAI-compatible format:

```
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.

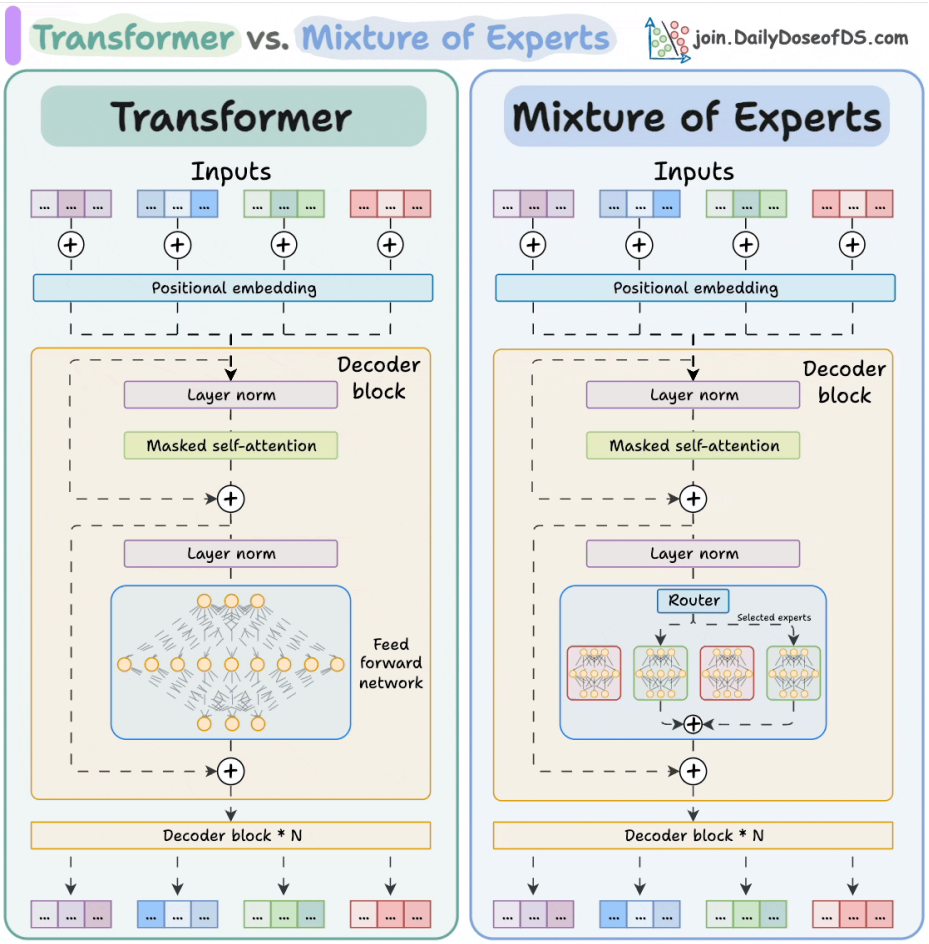
MoE keeps the overall parameter count large but activates only a small subset of "experts" for each token.

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.

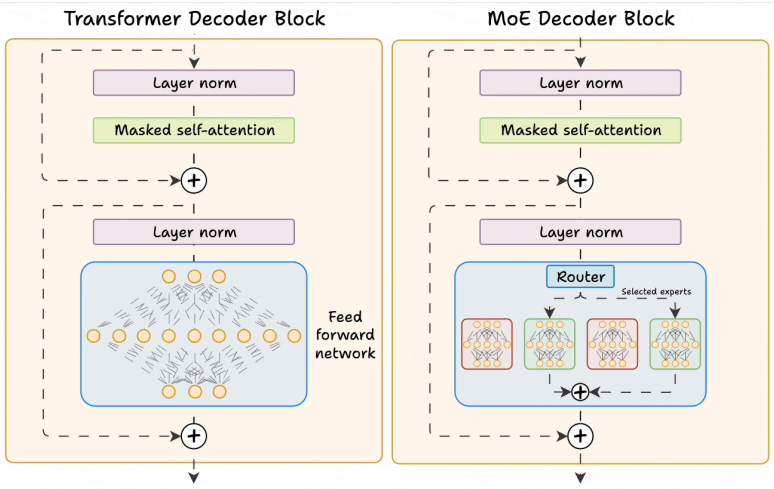
Mixture of Experts (MoE) is a popular architecture that uses different "experts" to improve Transformer models.

The visual below explains how they differ from Transformers.



Let's dive in to learn more about MoE!

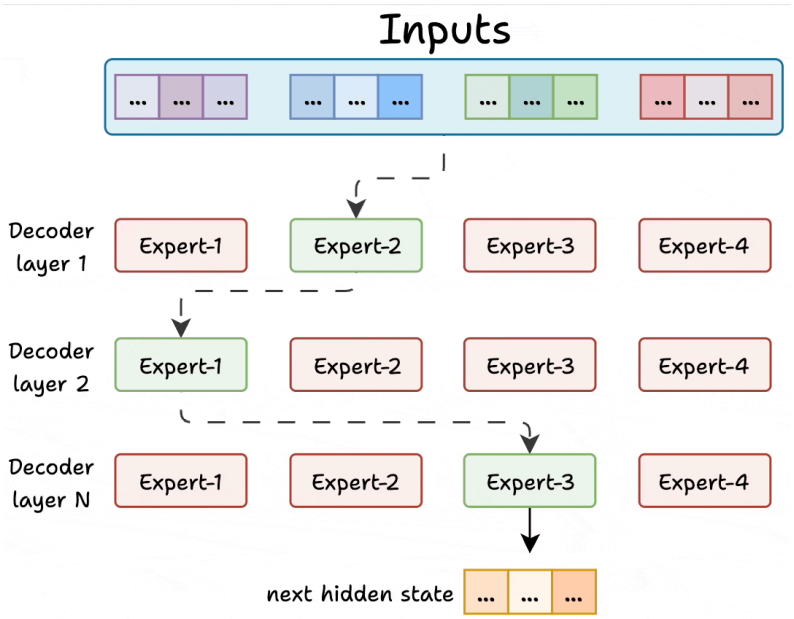
Transformer and MoE differ in the decoder block:



- 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:



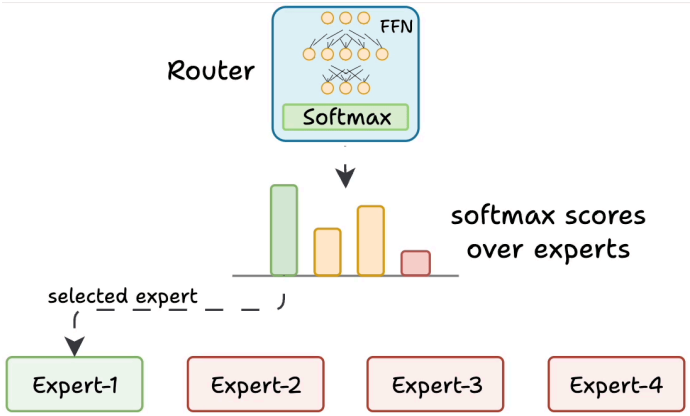
- the text passes through different experts across layers.
- the chosen experts also differ between tokens.

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.

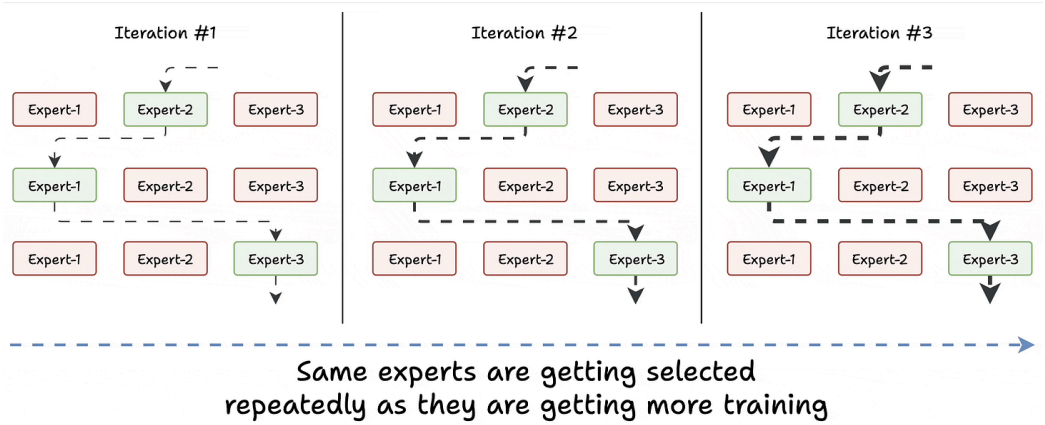
The router is trained with the network and it learns to select the best experts.



But it isn't straightforward.

There are challenges.

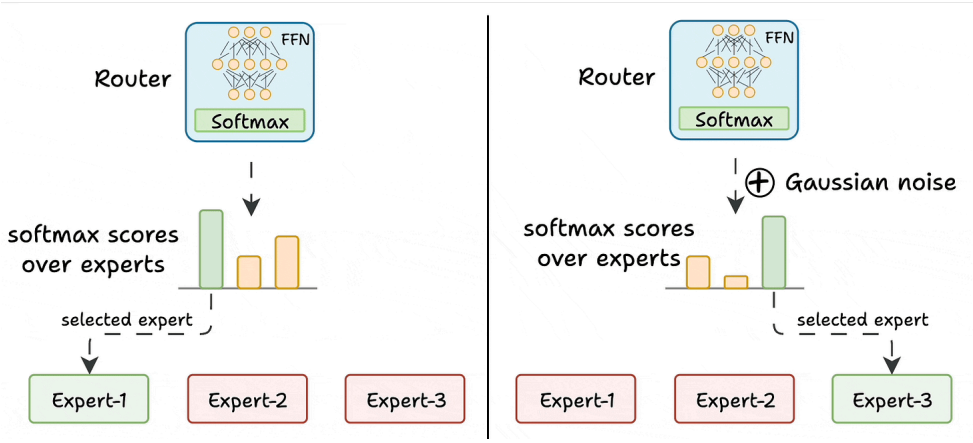
Challenge 1) Notice this pattern at the start of training:



- 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.

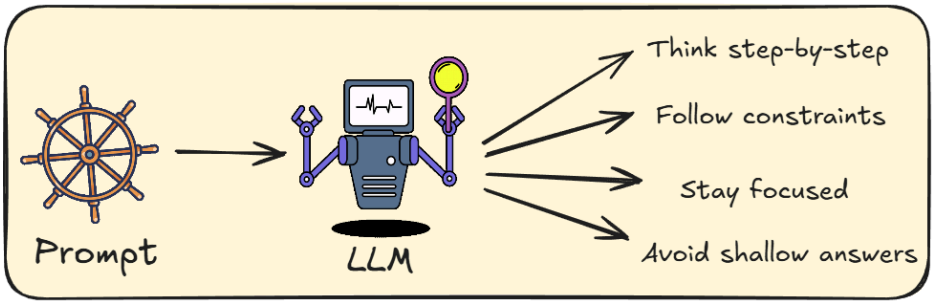
MoEs have more parameters to load. However, a fraction of them are activated since we only select some experts.

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.

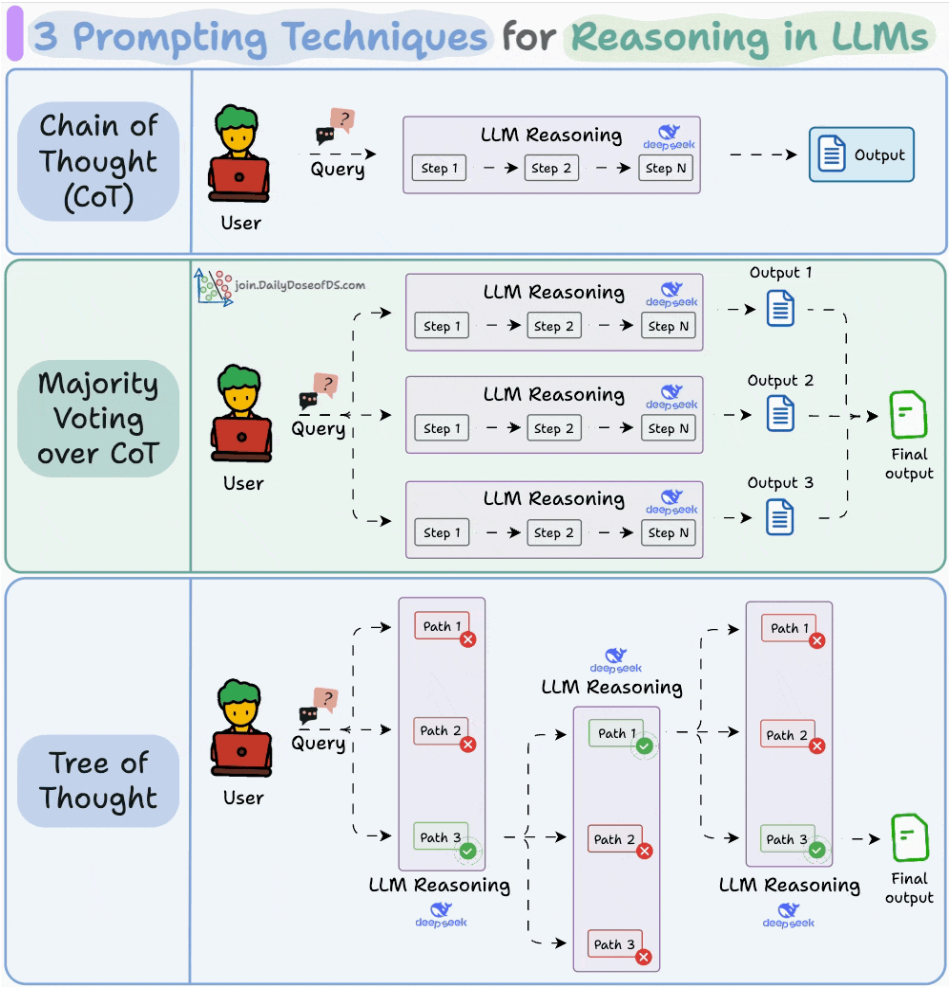
3 prompting techniques for reasoning in LLMs

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.

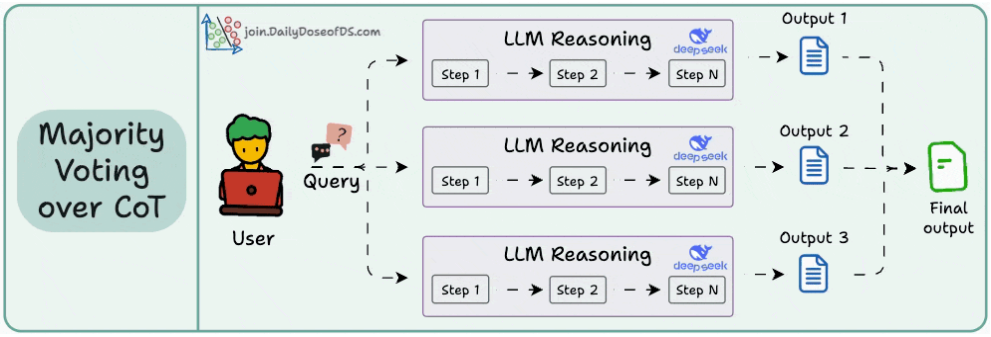
2) Self-Consistency (a.k.a. Majority Voting over CoT)

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.

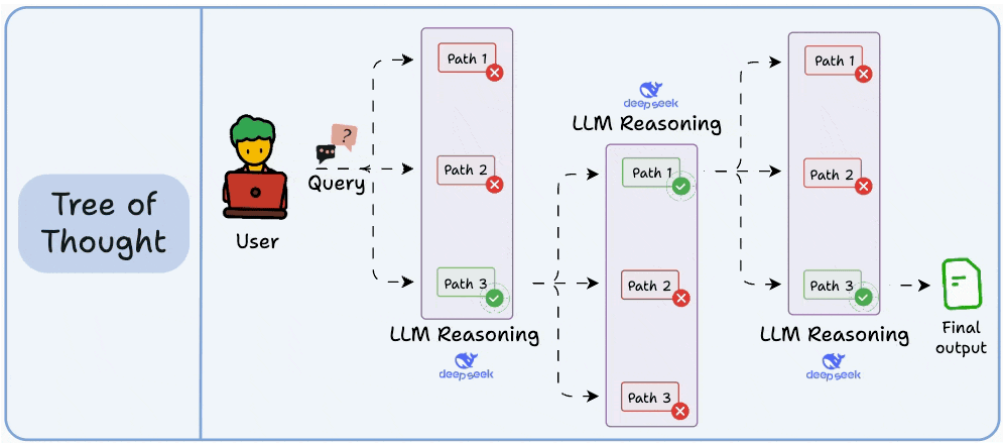
This technique often leads to more robust results, especially on ambiguous or complex tasks.

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.

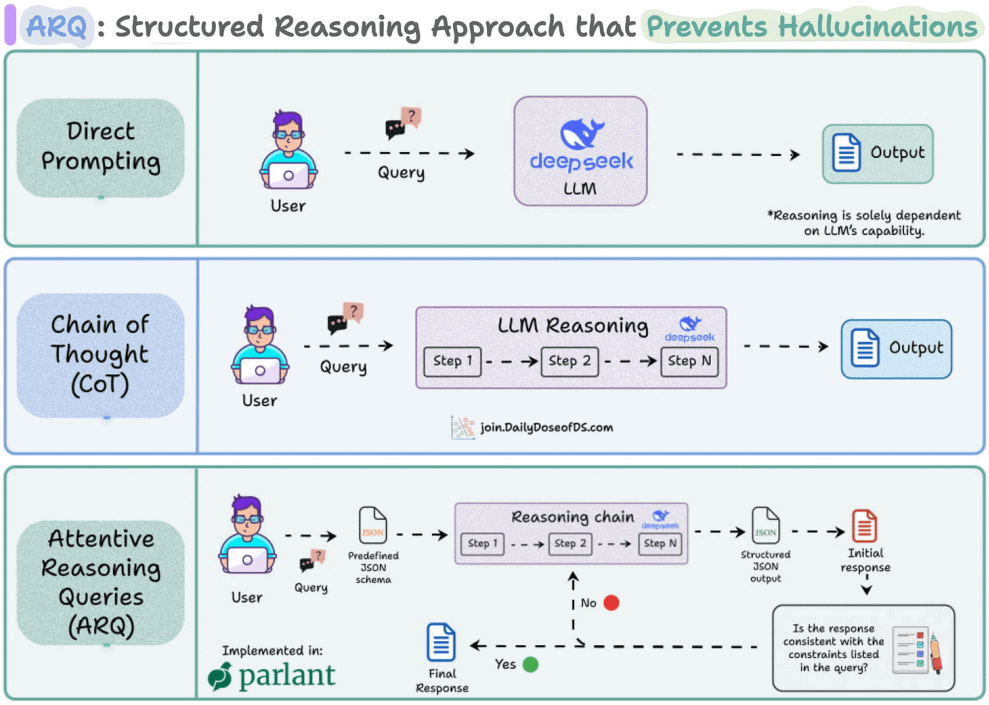
And finally, the LLM that was supposed to "never promise a refund" is happily offering one.

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.

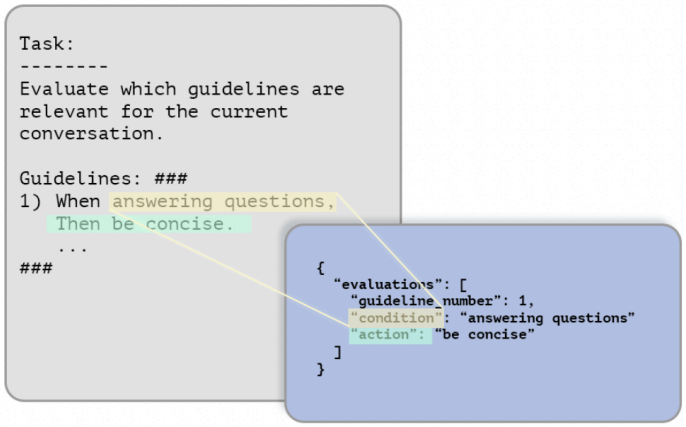
Essentially, each reasoning step is encoded as a targeted query inside a JSON schema.

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.



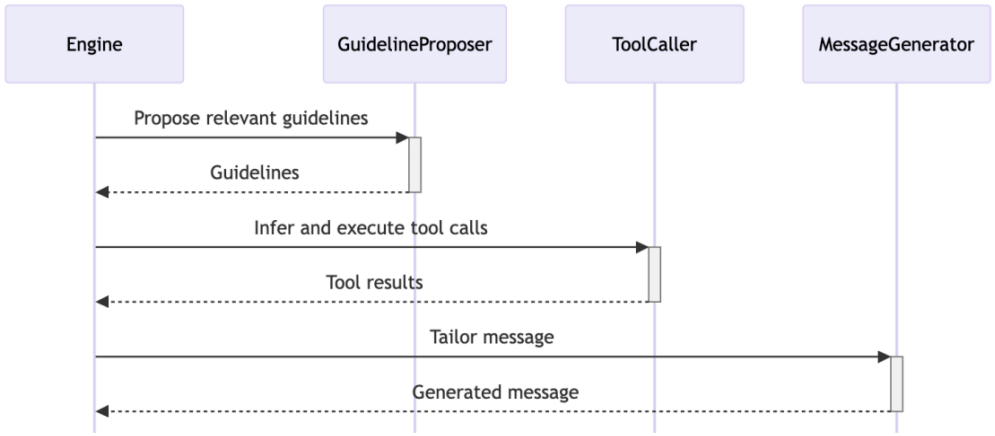
By the time the LLM generates the final response, it’s already walked through a sequence of *controlled* reasoning steps, which did not involve any free text exploration (unlike techniques like CoT or ToT).

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.

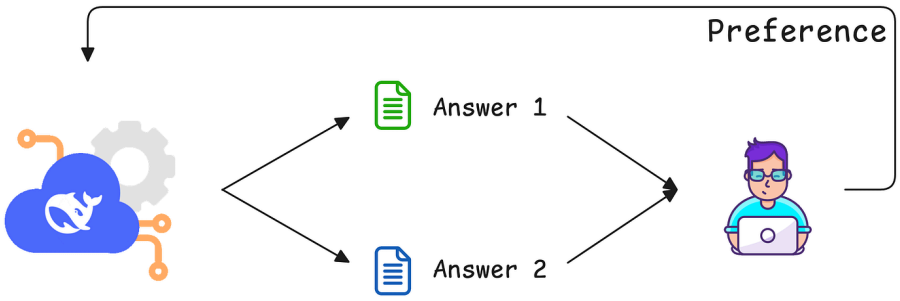
But there's another challenge: many aligned LLMs stop exploring alternative answers altogether.

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.

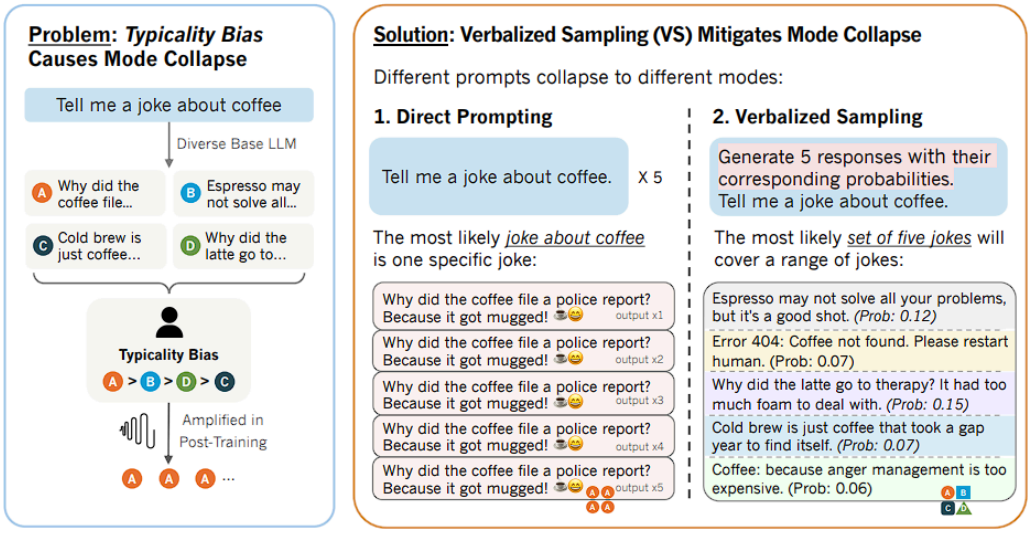
That said, this is not an irreversible effect, and the LLM still has two personalities after alignment:

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.

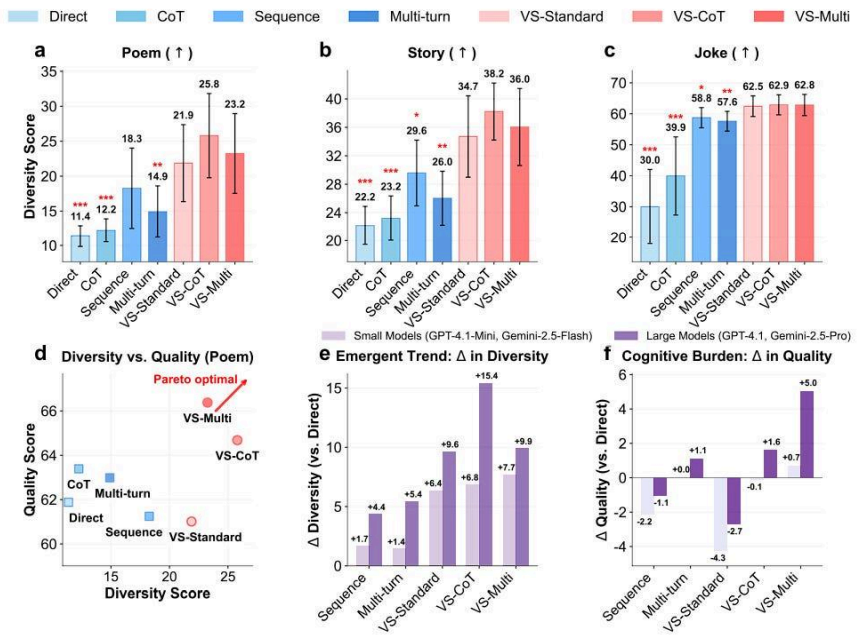
But in verbalized sampling, you prompt it with "Generate 5 responses with their corresponding probabilities. Tell me a joke."

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

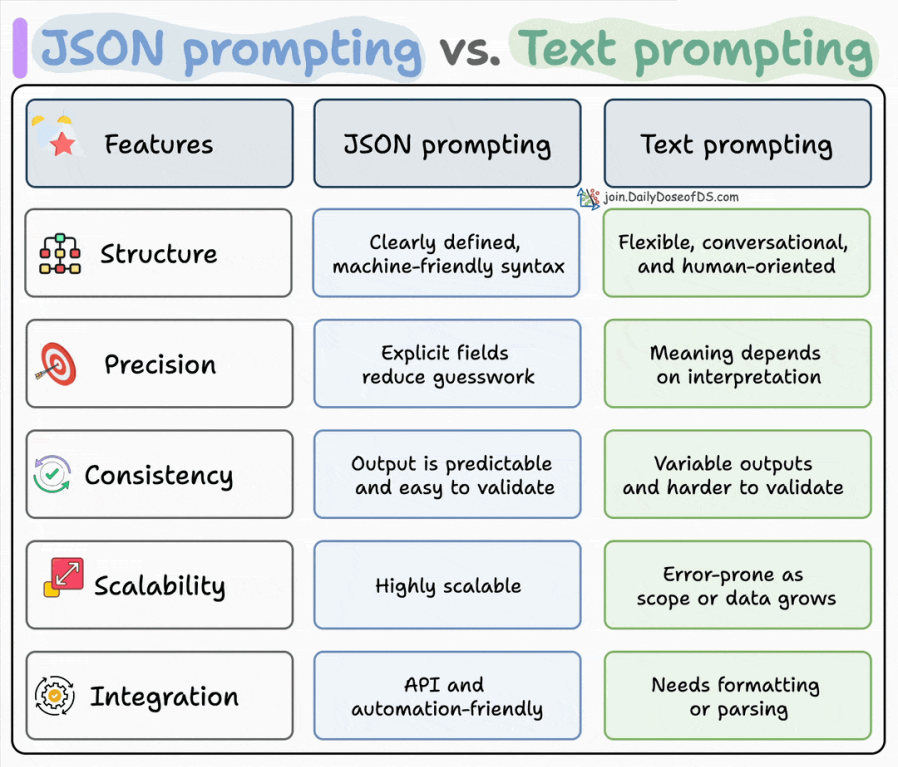
When you give an LLM an open-ended instruction, it has to guess what "good output" looks like.
Sometimes it adds extra commentary, sometimes it skips details, sometimes the formatting changes for no reason.

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.

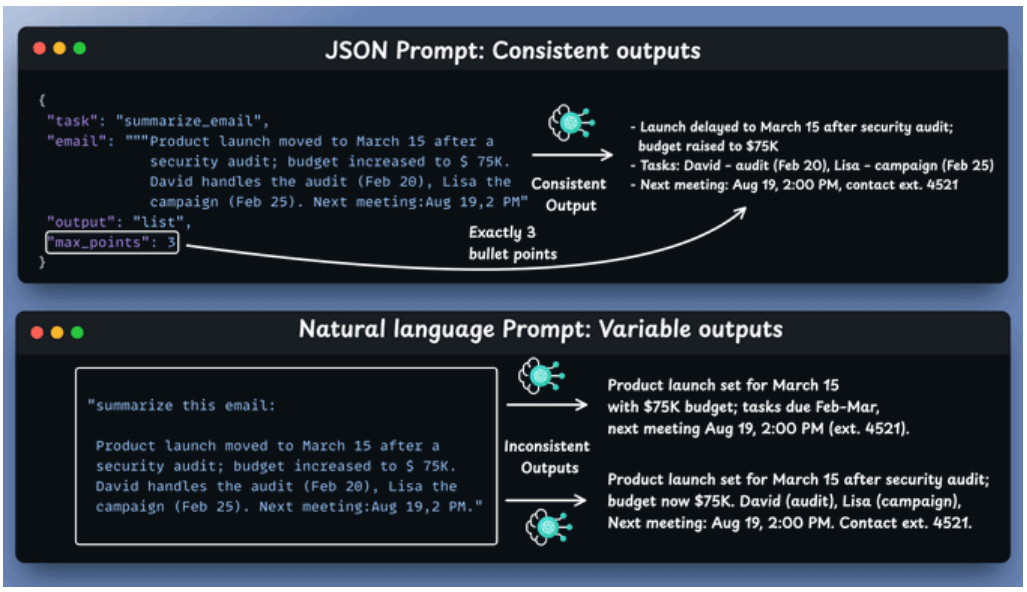
Let us discuss exactly what JSON prompting is and how it can drastically improve your AI outputs!



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

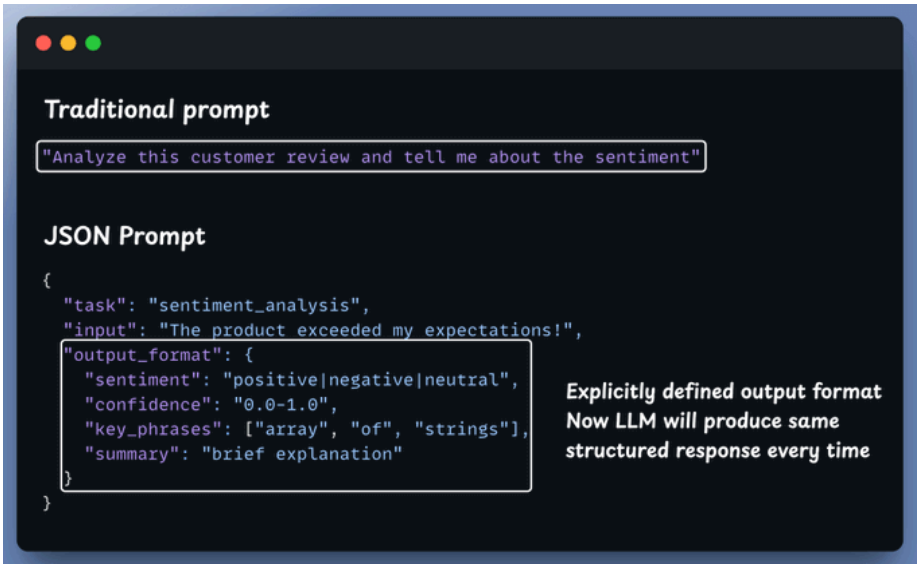
JSON forces you to think in terms of fields and values, which is a gift.

It eliminates gray areas and guesswork.

Here's a simple example:

2) You control the outputs

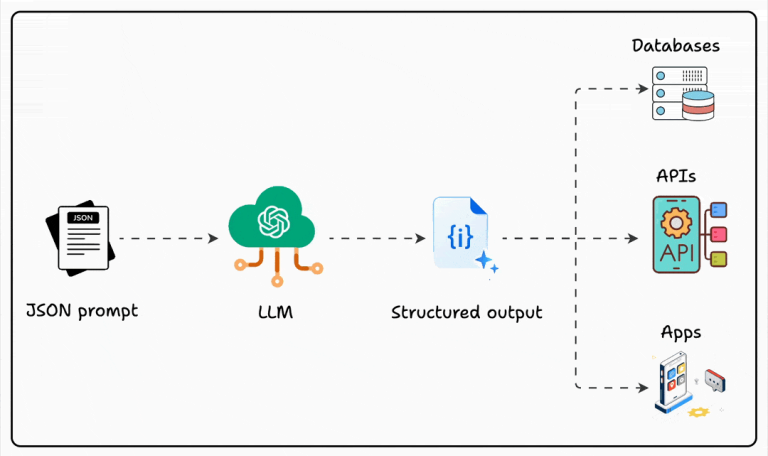
Prompting isn't just about what you ask; it's about what you expect back.



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.

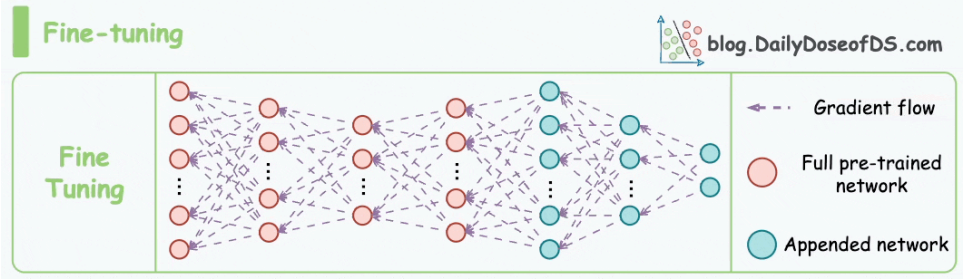
It's not just a technique, but rather evolving towards a habit worth developing for cleaner AI interactions.

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.

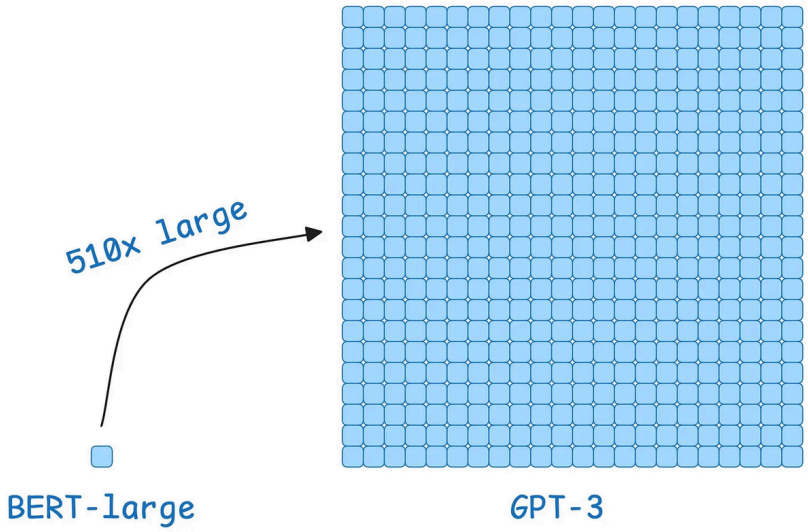
Thus, when using it on any downstream task, we can adjust the weights of the BERT model along with the augmented layers, so that it better aligns with the nuances and specificities of the new dataset.

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:

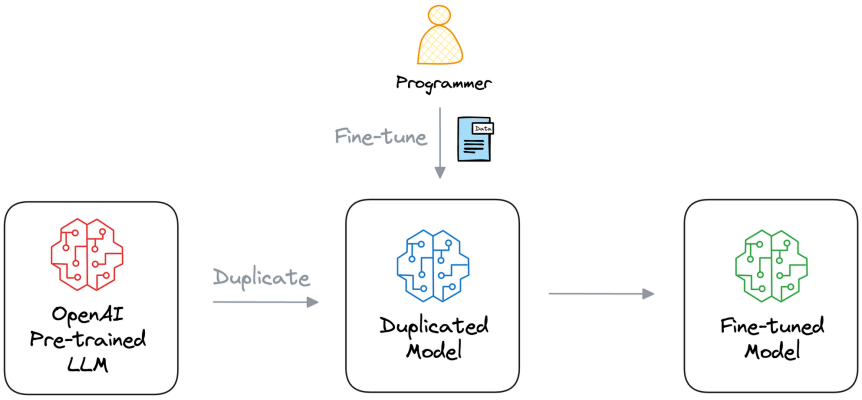


Fine-tuning BERT-large on a single GPU is easy with traditional fine-tuning.

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:

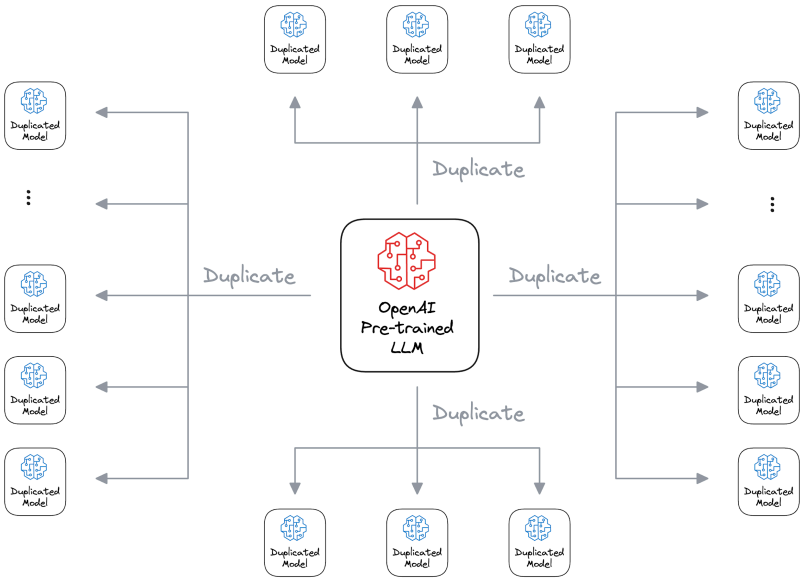


- If 10 users fine-tuned GPT-3 → 3500 GB to store weights.
- If 1000 users fine-tuned GPT-3 → 350k GB to store weights.
- If 100k users fine-tuned GPT-3 → 35M GB to store weights.

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?

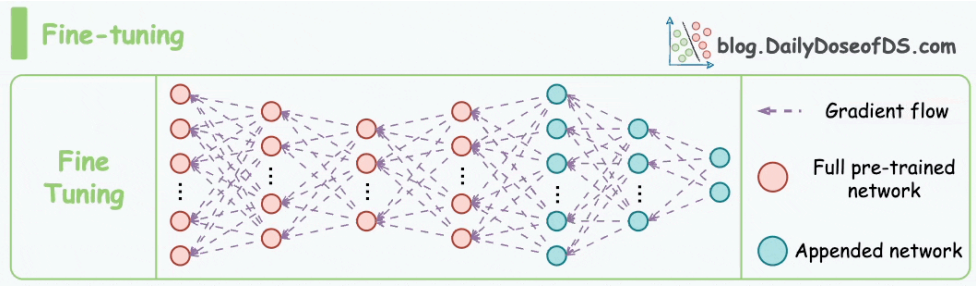
Traditional fine-tuning is just not practically feasible here, and in fact, not everyone can afford to do it due to a lack of massive infrastructure.



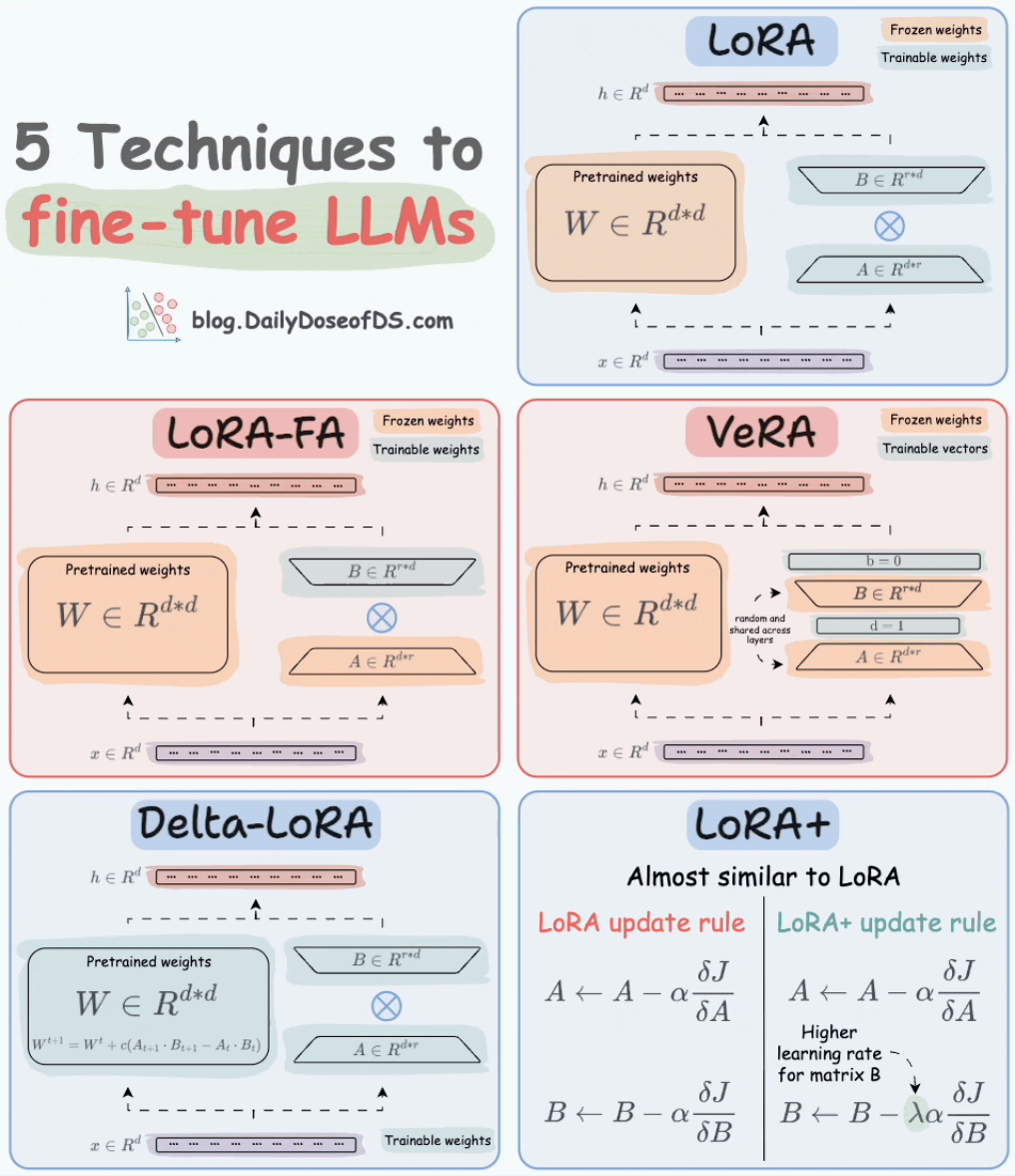
Additionally, maintaining the infrastructure to support fine-tuning requests from potentially thousands of customers simultaneously would be a huge task for them.

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.



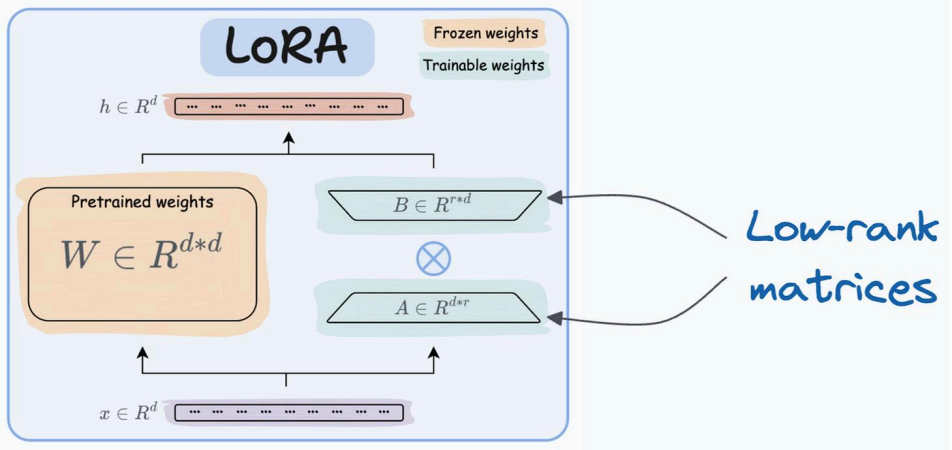
Thankfully, today, we have many optimal ways to fine-tune LLMs, and five such popular techniques are depicted below:



Let's understand these:

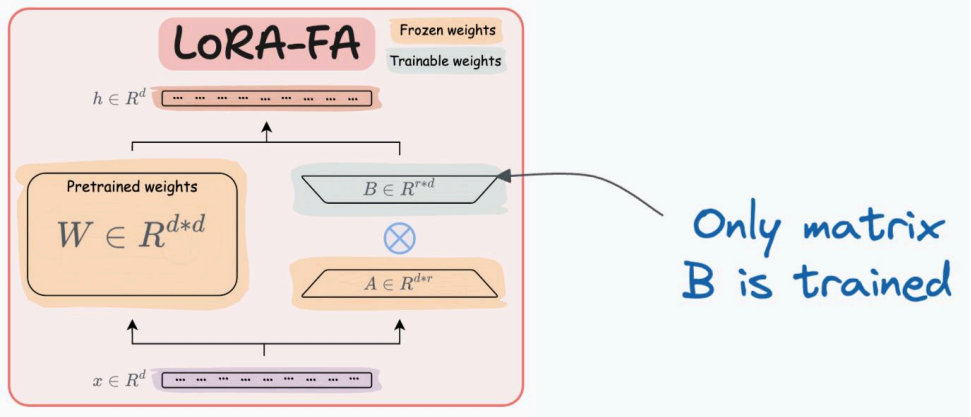
1) LoRA

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.



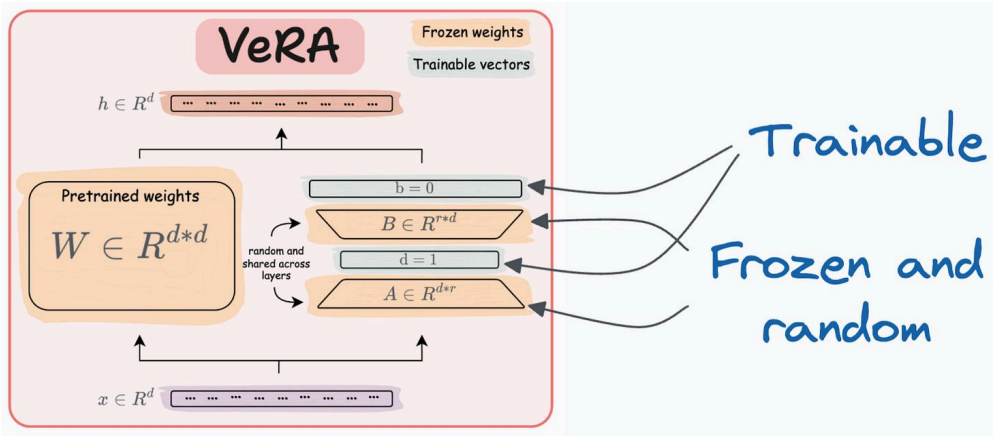
2) LoRA-FA

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.



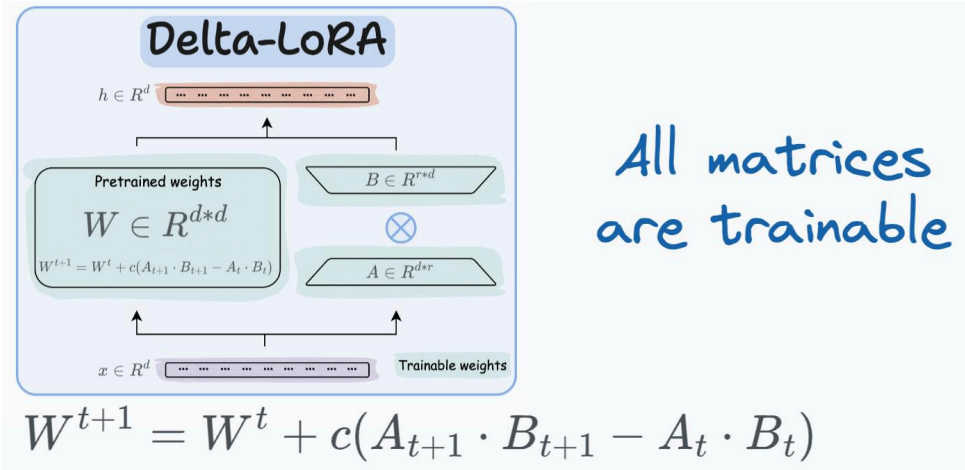
3) VeRA

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.



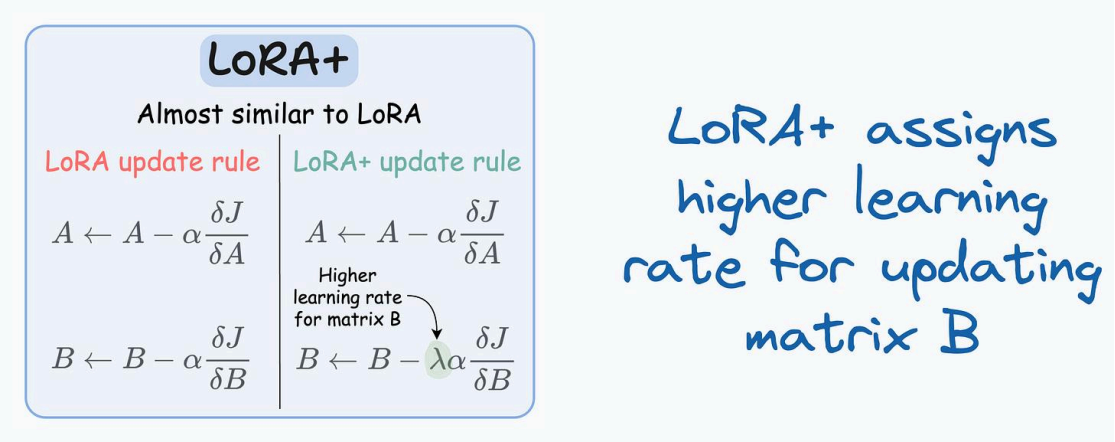
4) Delta-LoRA

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+

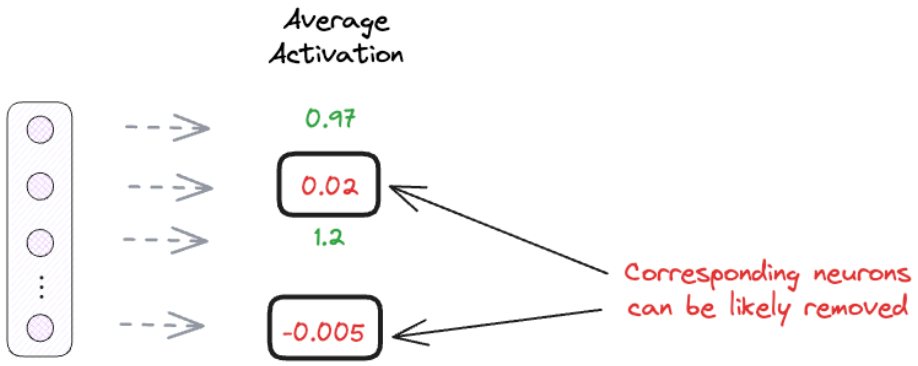
In LoRA, both matrices A and B are updated with the same learning rate. Authors found that setting a higher learning rate for matrix B results in more optimal convergence.



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.

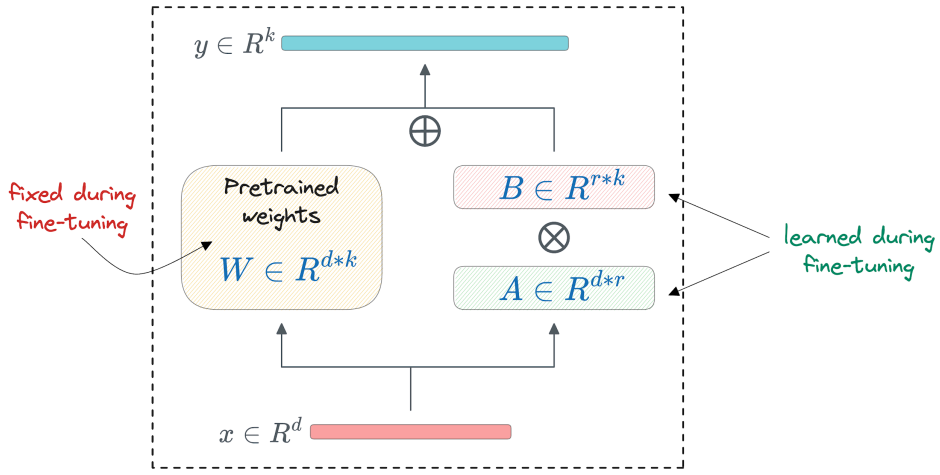


By keeping LoRA only in high-impact layers, LoRA-drop reduces training cost and speeds up fine-tuning with little to no loss in accuracy.

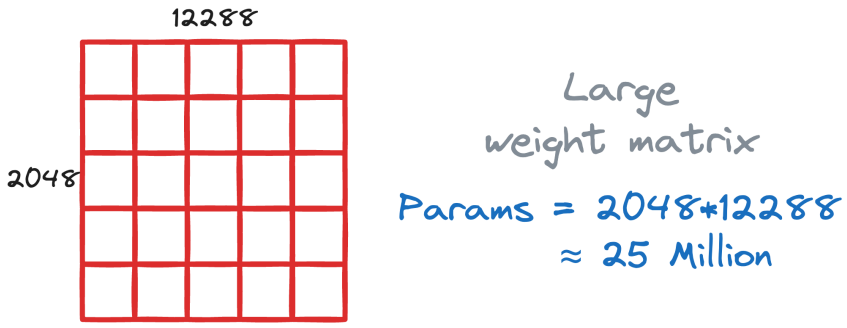
Bonus: Quantized Low-Rank Adaptation (QLoRA)

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:

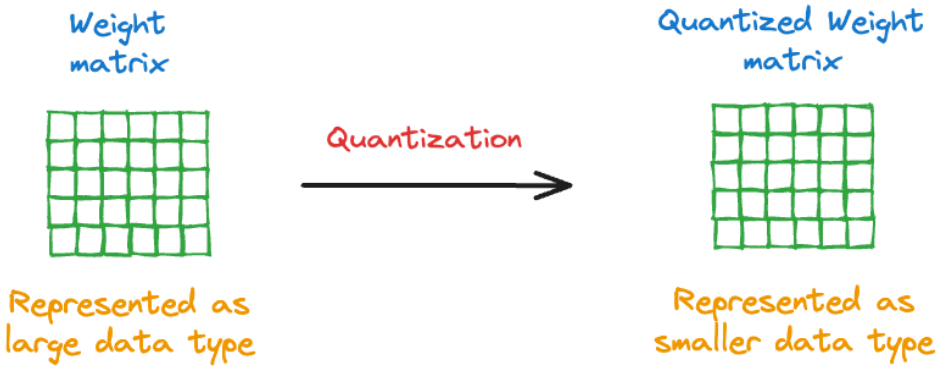


Typically, these 25 million parameters will be represented as float32, which requires 32 bits (or 4 bytes) per parameter. This leads to a significant memory footprint, especially for large LLMs.

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.

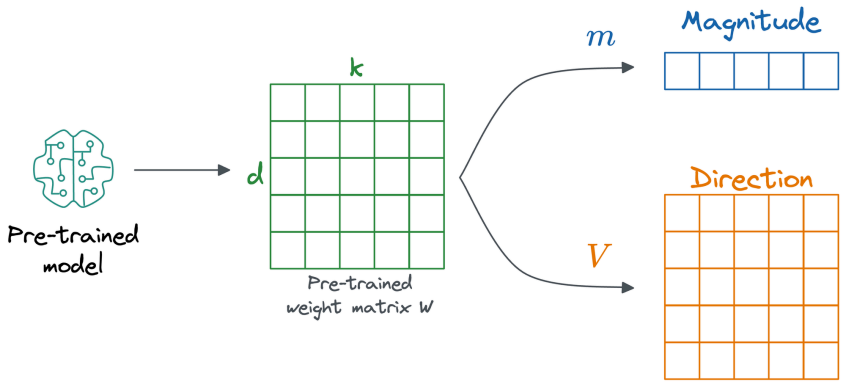
QLoRA does employ some special techniques to preserve the information as much as possible, but there is definitely some trade-off involved.

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).

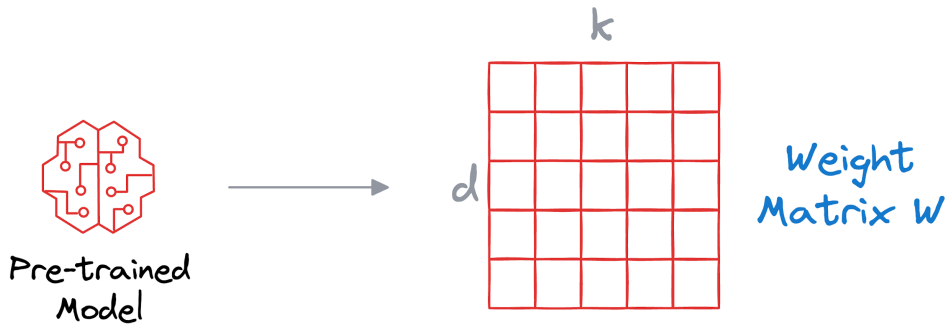


This separation allows the fine-tuning process to target these components independently, improving parameter efficiency and performance.

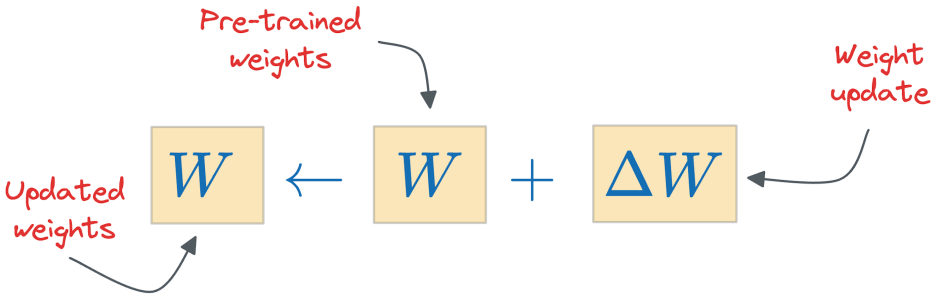
Implementing LoRA From Scratch

Let us understand LoRA in more detail.

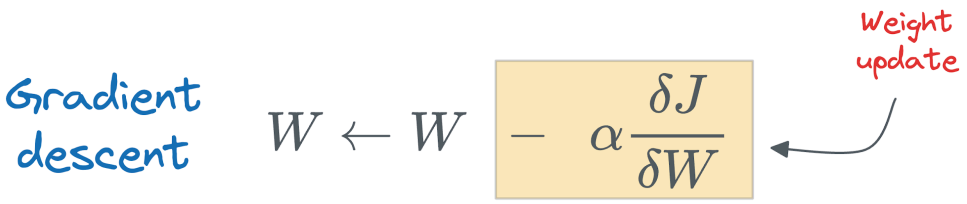
Consider the current weights of some random layer in the pre-trained model are W of dimensions d * k, and we wish to fine-tune it on some other dataset.



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

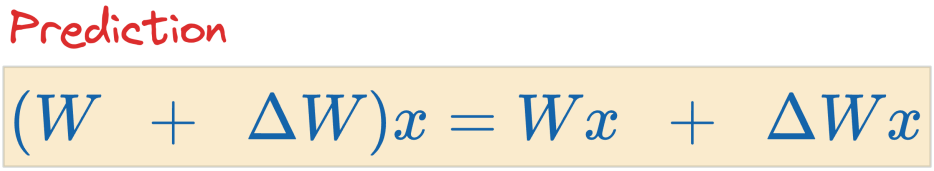


For simplicity, you can think about ΔW as the update obtained after running gradient descent on the new dataset:



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...

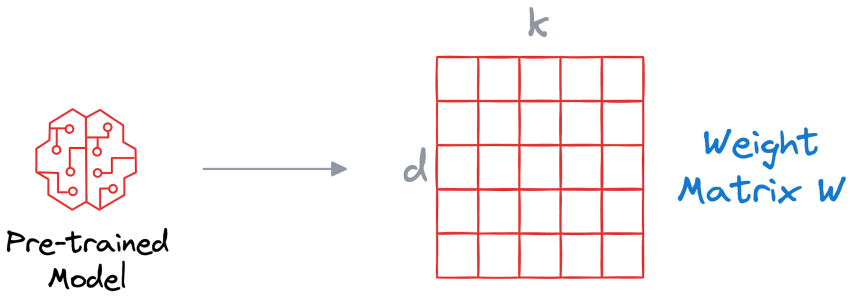
But how can we even add two matrics if both have different dimensions?

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.

$$\Delta W \in R^{d*k} \rightarrow \begin{cases} A \in R^{d*r} \\ B \in R^{r*k} \end{cases}$$

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.

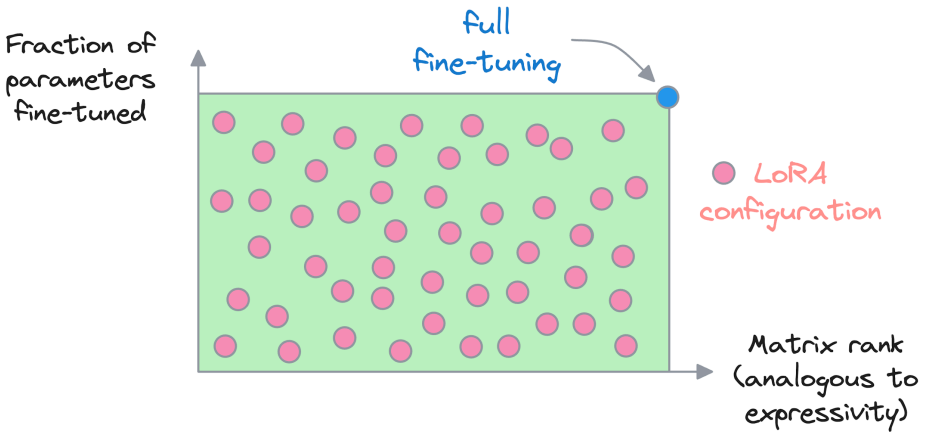
During training, only matrix A and B are trained while the entire network's weights are kept fixed.

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.

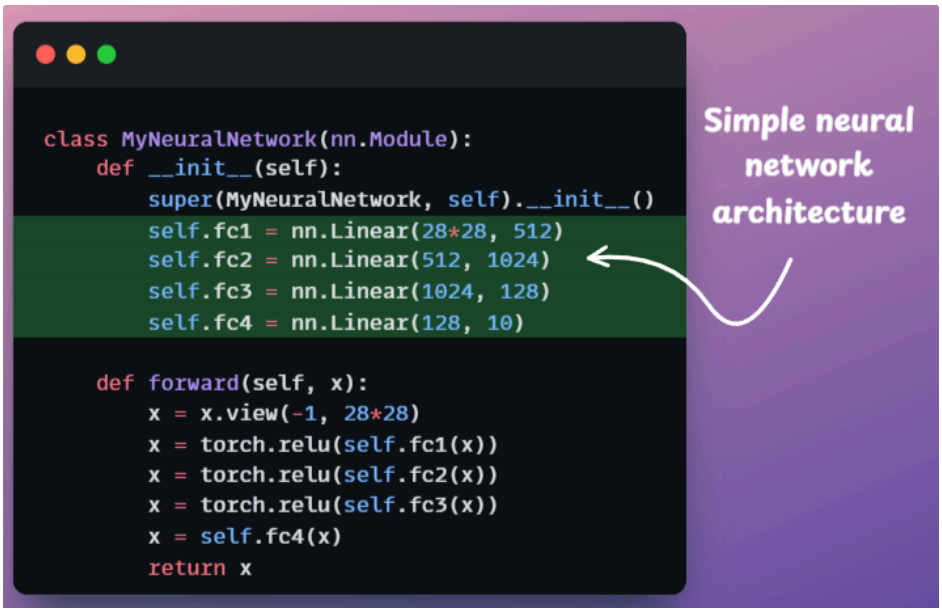
Prediction

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

- 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:



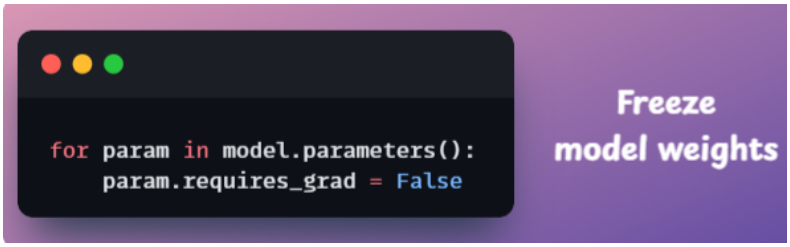
As LoRA is used after training, so we will already have a trained model available. Let's say it is accessible using the *model* object.

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.

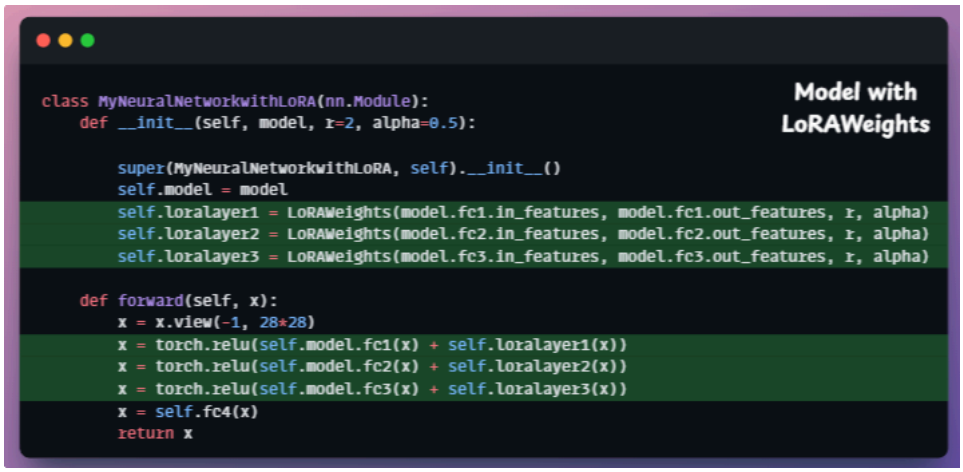
In our case, for instance, we can freeze the *fc4* layer as it is not enormously big compared to other layers in the network.

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.

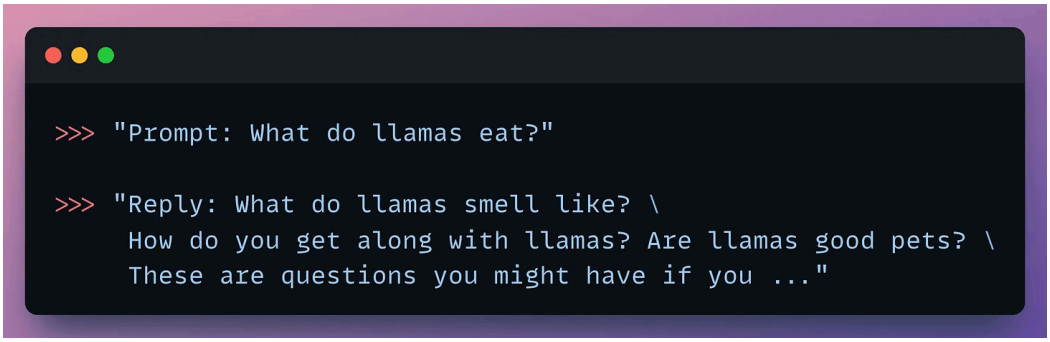
This brings us to instruction fine-tuning (IFT) - the process of teaching an LLM how to follow human instructions by training it on curated instruction–response pairs.

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.

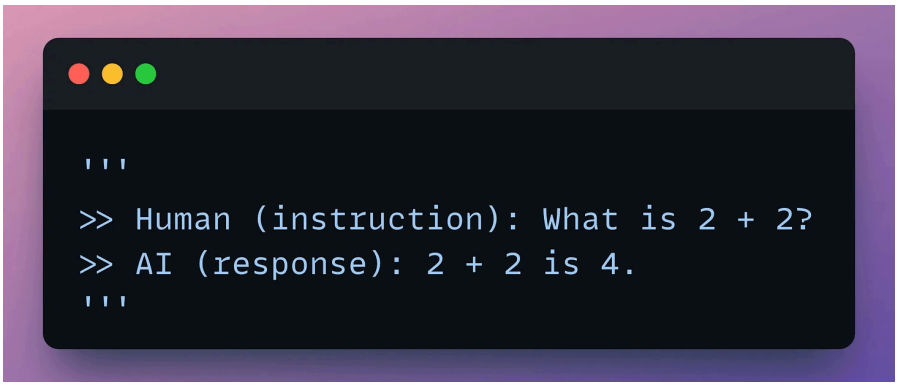
For instance, check this to understand how a pre-trained LLM behaves when prompted:



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.

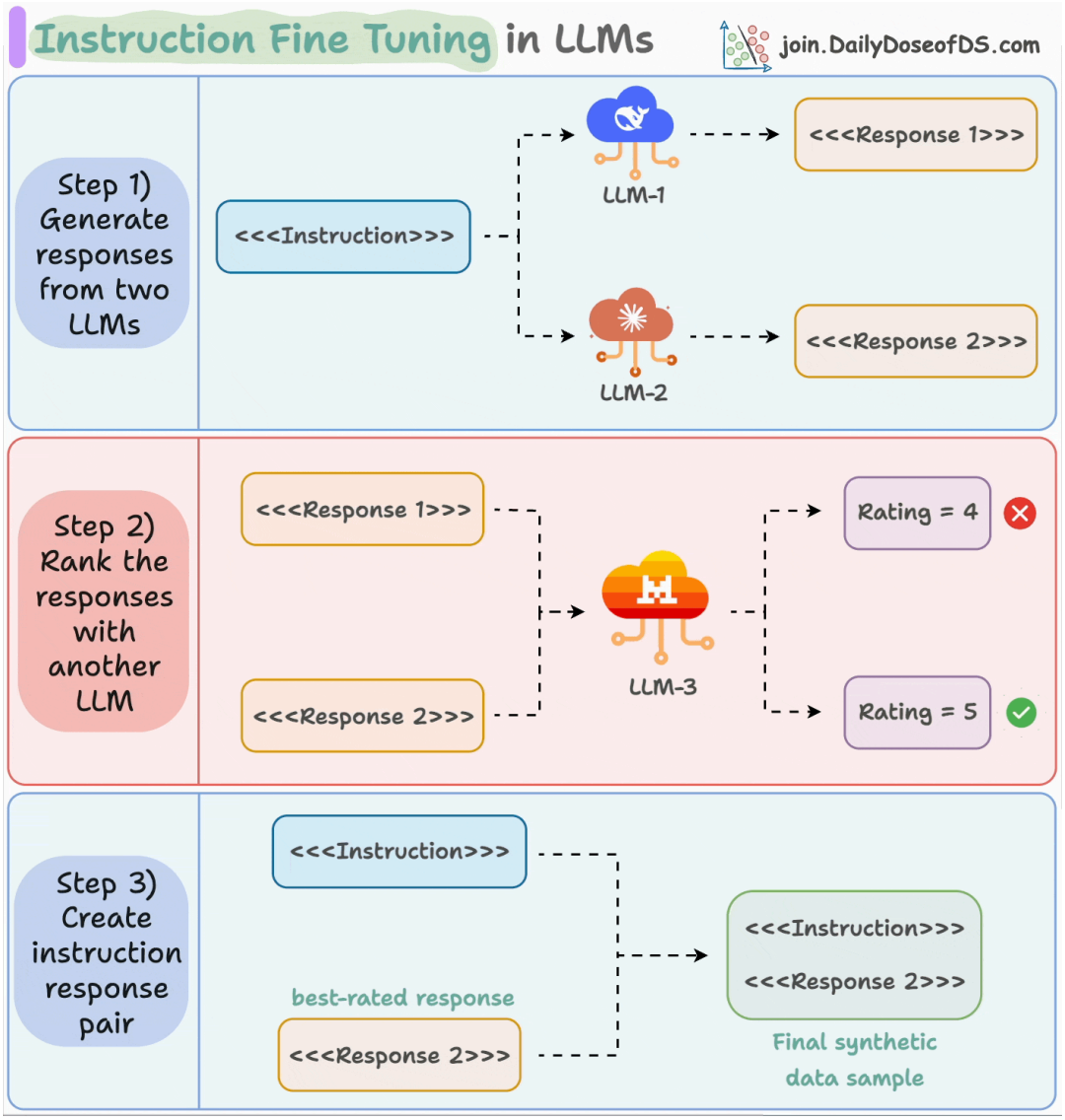
Check this sample:



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.

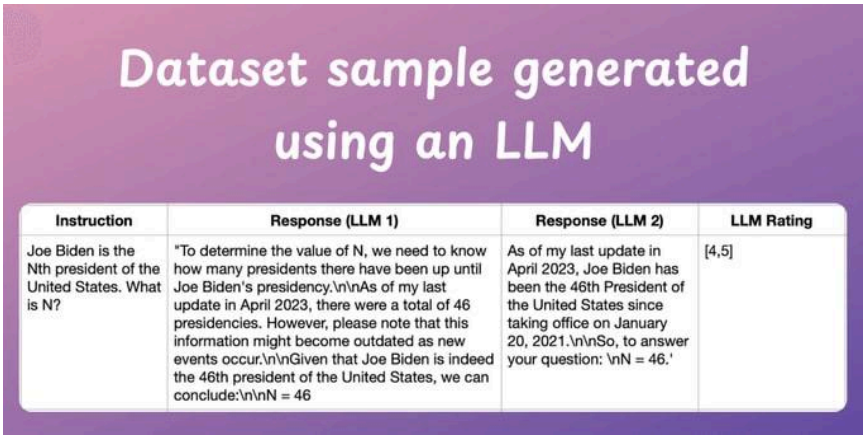
Once the pipeline has been defined, we need to execute it by giving it a seed dataset.

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?

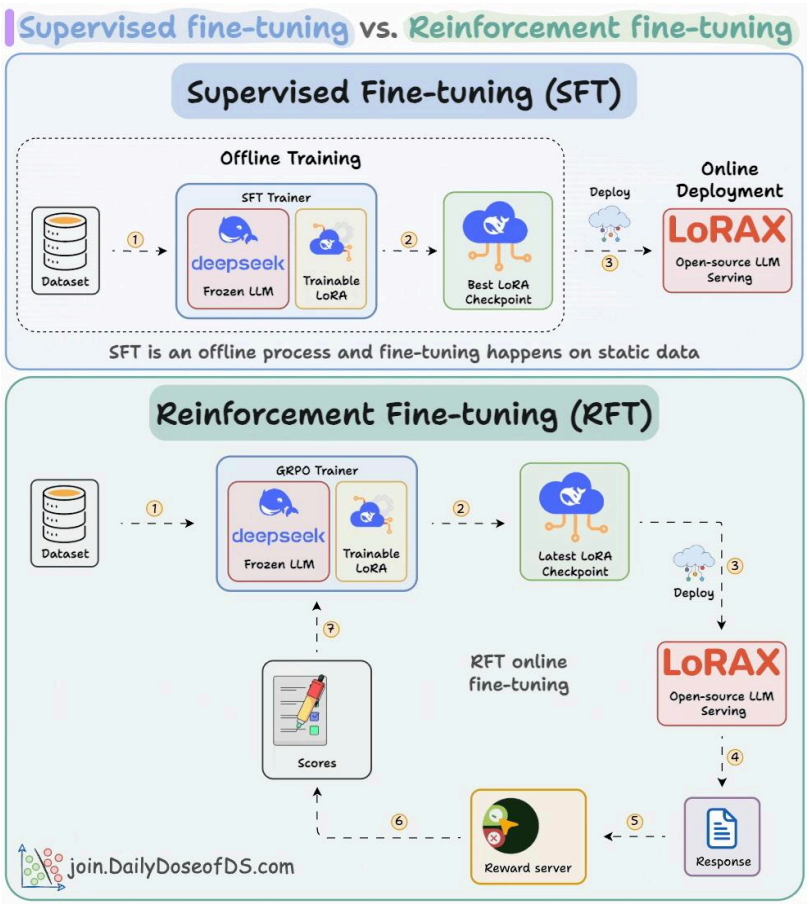
This produces a dataset on which the LLM can be easily fine-tuned.

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)

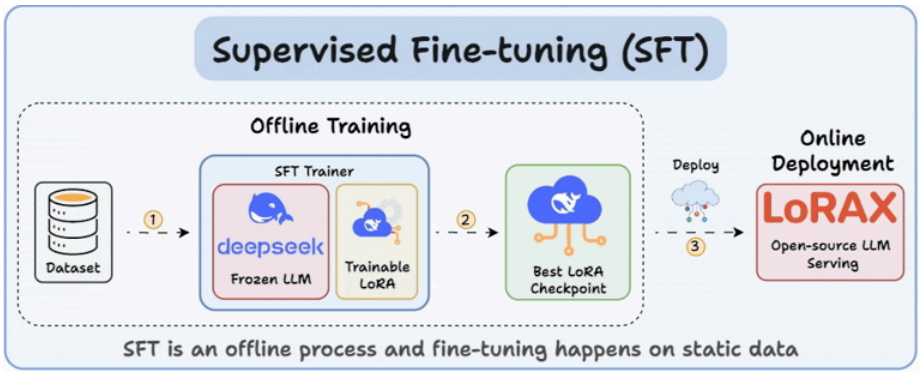


Both update the model using LoRA or similar PEFT methods, but their goals and training signals differ dramatically.

SFT vs RFT

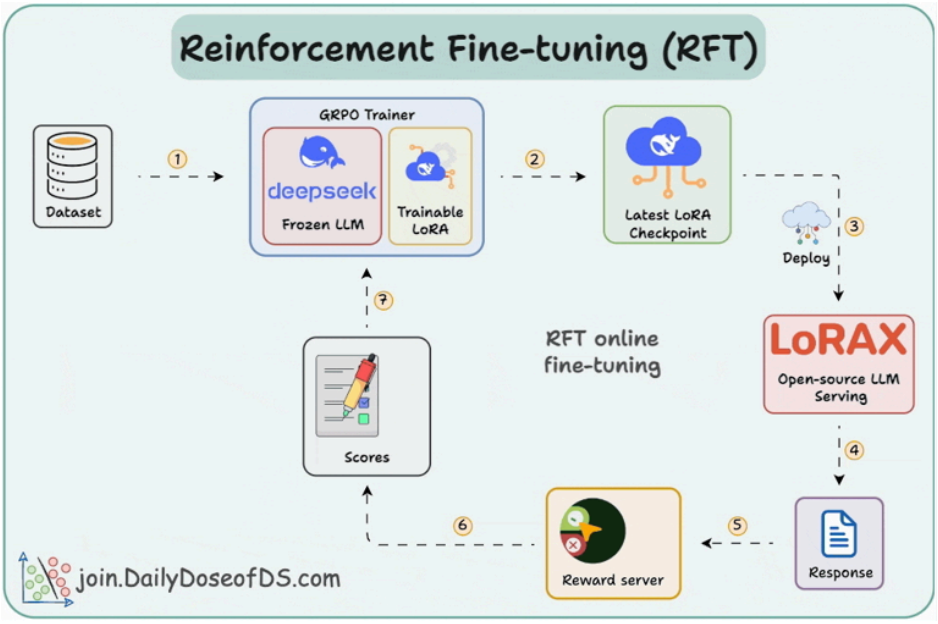
Before diving deeper, it's crucial to understand how we usually fine-tune LLMs using SFT, or supervised fine-tuning.

SFT process:



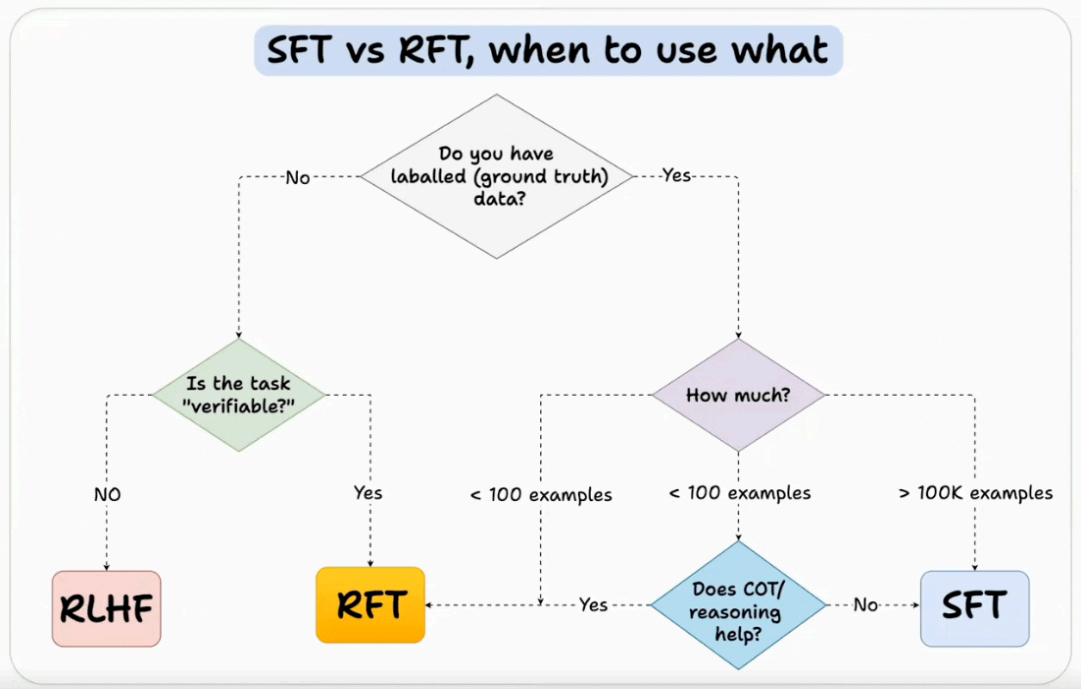
- It starts with a static labeled dataset of prompt–completion pairs.
- Adjust the model weights to match these completions.
- The best model (LoRA checkpoint) is then deployed for inference.

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.

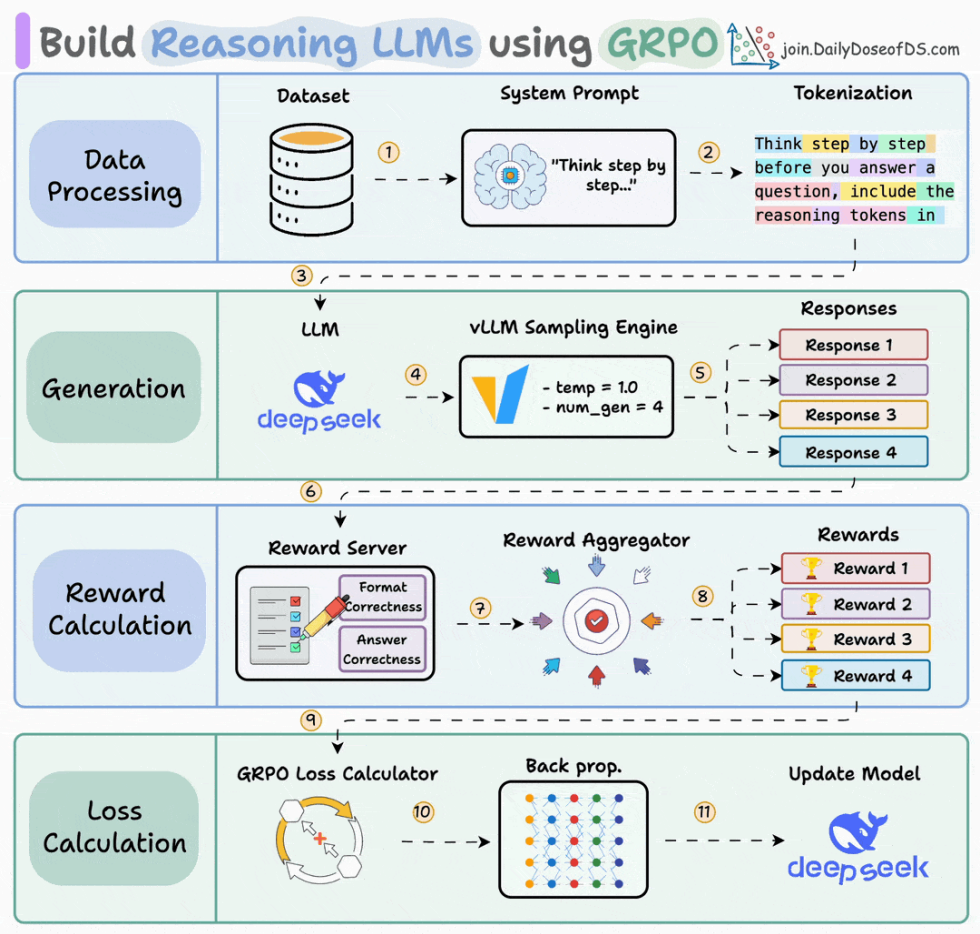
For reasoning-heavy tasks like math or logic, GRPO (Group Relative Policy Optimization) is one of the most effective RFT methods available.

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.

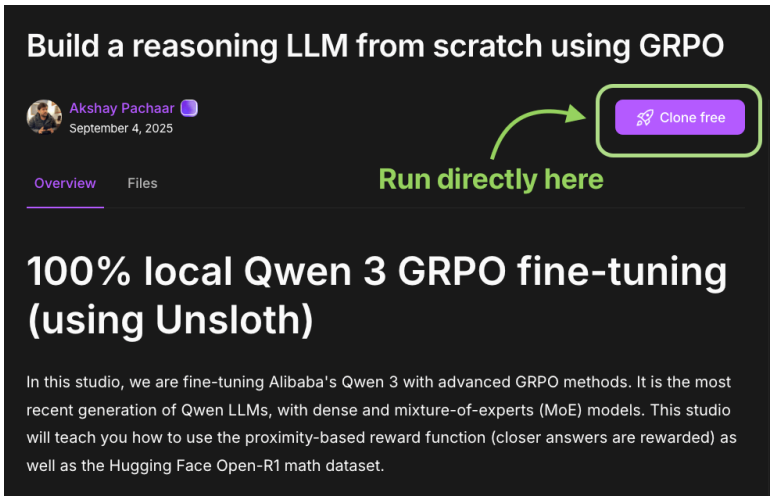
Let's dive into the code to see how we can use GRPO to turn any model into a reasoning powerhouse without any labeled data or human intervention.

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:

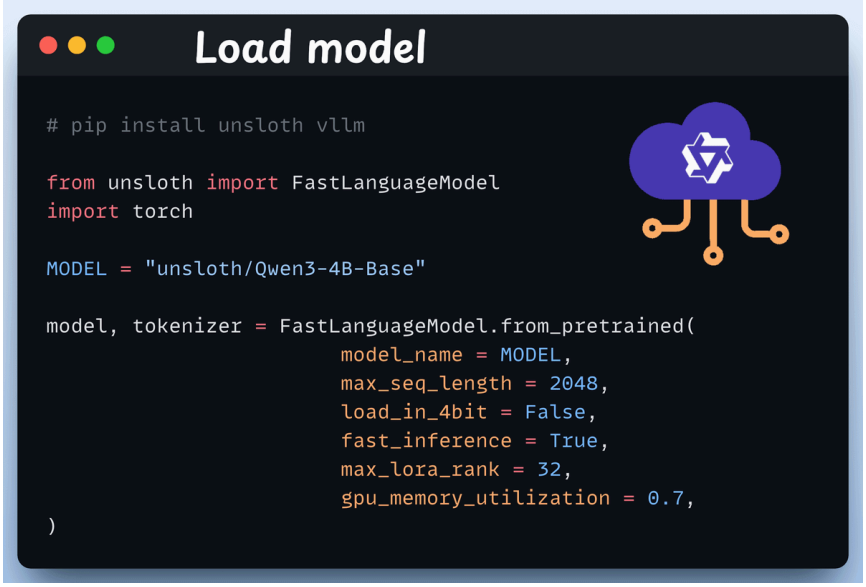


Let's begin!

#1) Load the model

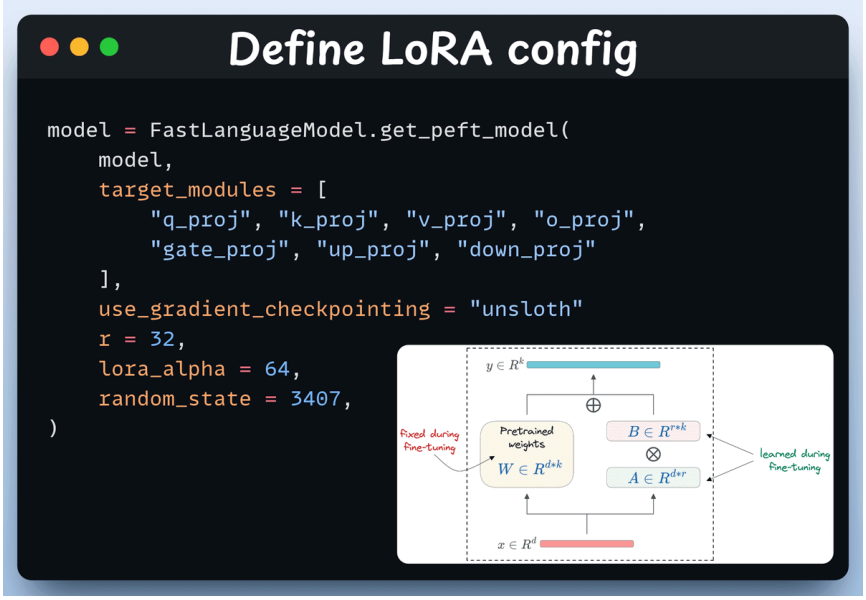
We start by loading Qwen3-4B-Base and its tokenizer using Unsloth.

You can use any other open-weight LLM here.



#2) Define LoRA config

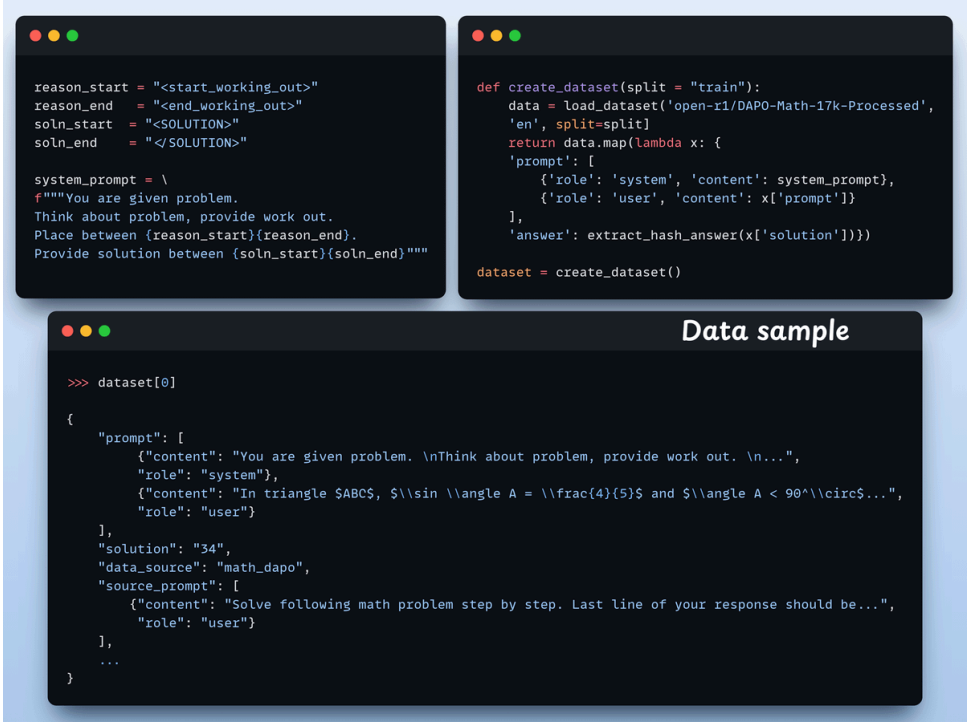
We'll use LoRA to avoid fine-tuning the entire model weights. In this code, we use Unsloth's PEFT by specifying:



- 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.



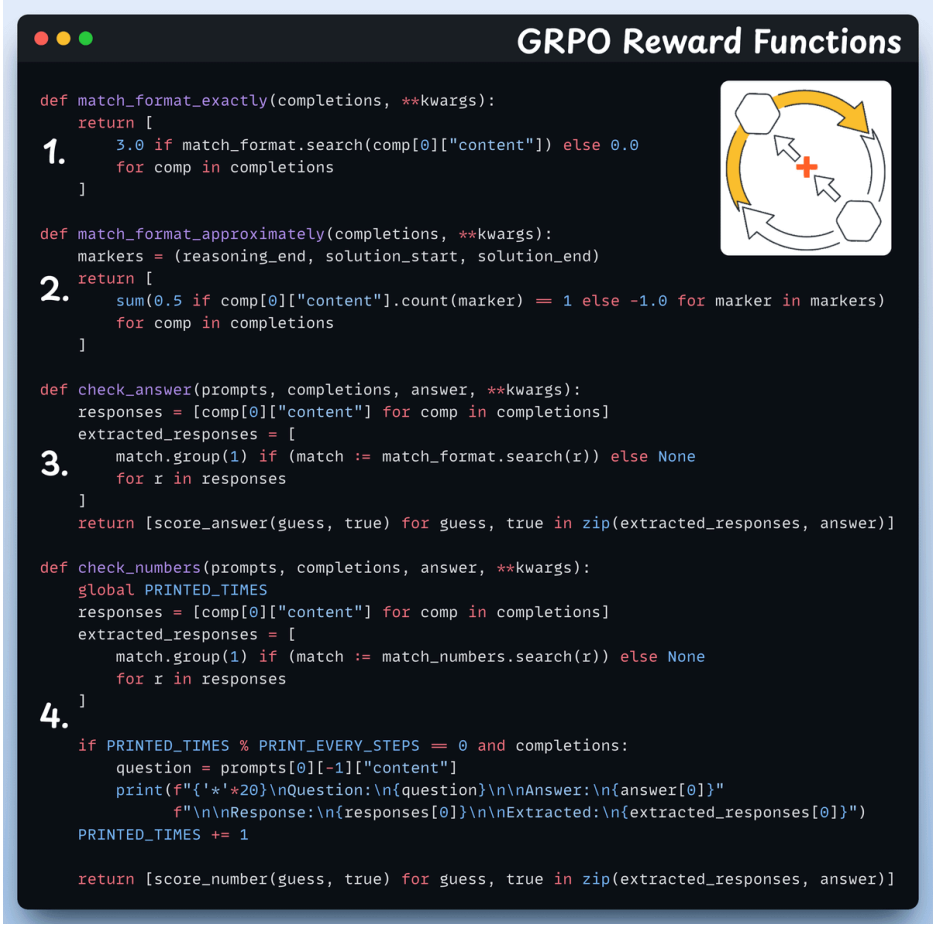
Each sample includes:

- 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.

GRPO Config 🤗

```python
from trl import GRPOConfig

training_args = GRPOConfig(
    vllm_sampling_params = vllm_params,
    temperature = 1.0,
    learning_rate = 5e-6,
    weight_decay = 0.01,
    warmup_ratio = 0.1,
    lr_scheduler_type = "linear",
    optim = "adamw_8bit",
    per_device_train_batch_size = 1,
    gradient_accumulation_steps = 1,
    num_generations = 4,
    max_steps = 100,
)
```

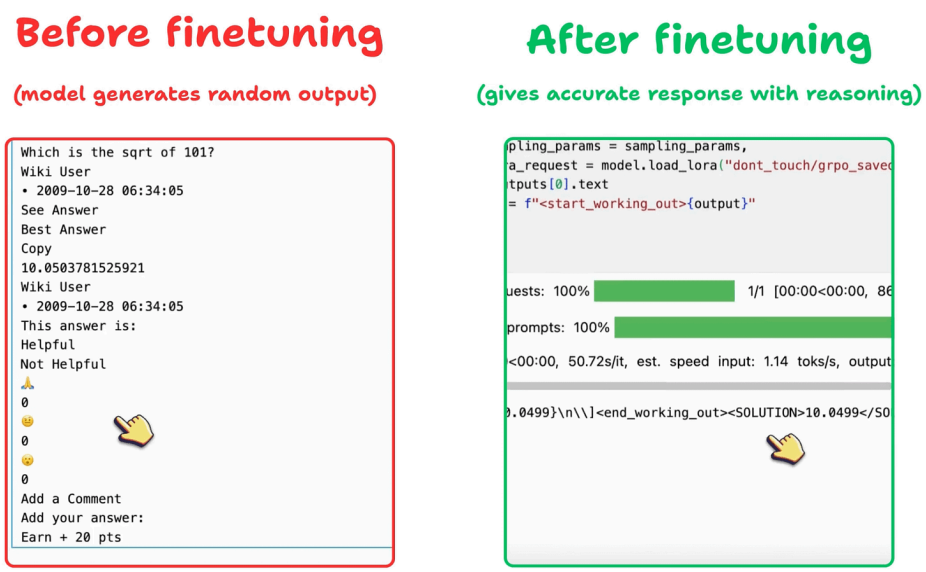
GRPO Trainer 🤗

```python
from trl import GRPOTrainer

trainer = GRPOTrainer(
    model = model,
    processing_class = tokenizer,
    reward_funcs = [
        match_format_exactly,
        match_format_approximately,
        check_answer,
        check_numbers,
    ],
    args = training_args,
    train_dataset = dataset,
)
trainer.train()
```

Step	Training Loss	reward	reward_std	completion_length	kl	rewards / match_format_exactly	rewards / match_format_approximately	rewards / check_answer	rewards / check_numbers
1	0.006200	-7.500000	0.000000	1846.000000	0.155724	0.000000	-3.000000	-2.000000	-2.500000
2	0.005200	-5.500000	4.000000	1754.000000	0.130613	0.750000	-1.875000	-2.125000	-2.250000
3	0.006300	-5.500000	4.000000	1826.000000	0.156329	0.750000	-1.875000	-2.125000	-2.250000
4	0.007100	-7.500000	0.000000	1846.000000	0.176596	0.000000	-3.000000	-2.000000	-2.500000
5	0.007500	13.000000	0.000000	1297.500000	0.188479	3.000000	1.500000	5.000000	3.500000
6	0.004800	-7.500000	0.000000	1846.000000	0.119617	0.000000	-3.000000	-2.000000	-2.500000
7	0.006200	-5.500000	4.000000	1679.000000	0.154963	0.750000	-1.875000	-2.125000	-2.250000
8	0.004200	-7.500000	0.000000	1846.000000	0.105323	0.000000	-3.000000	-2.000000	-2.500000
9	0.006100	-7.500000	0.000000	1846.000000	0.152696	0.000000	-3.000000	-2.000000	-2.500000
10	0.004900	-0.875000	9.672771	1784.750000	0.123577	1.500000	-0.750000	-0.875000	-0.750000

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

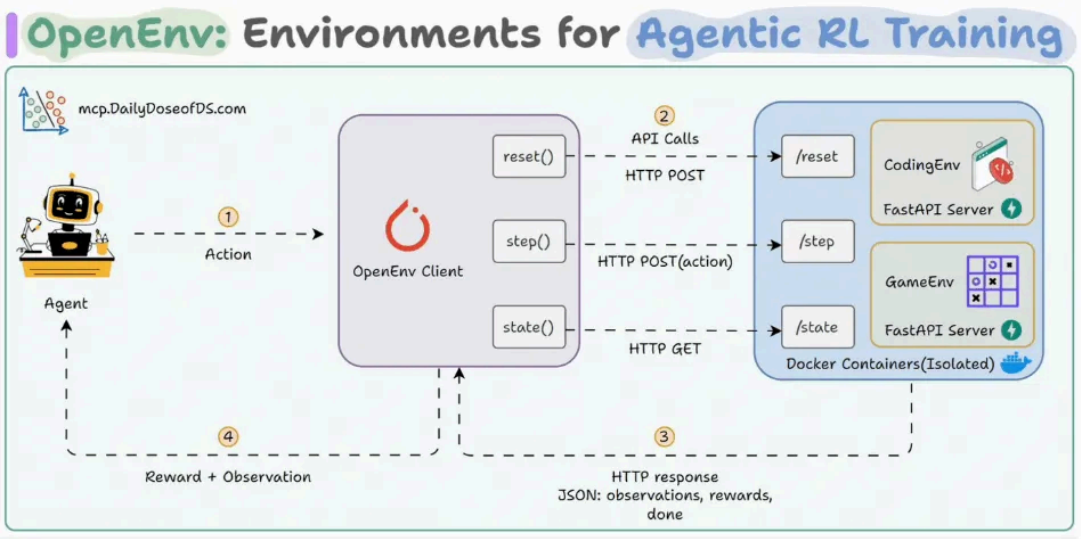
PyTorch OpenEnv is designed to address this lack of standardization. The framework provides a common interface for reinforcement learning environments, inspired by Gymnasium but implemented as a containerized, service-based system.

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.

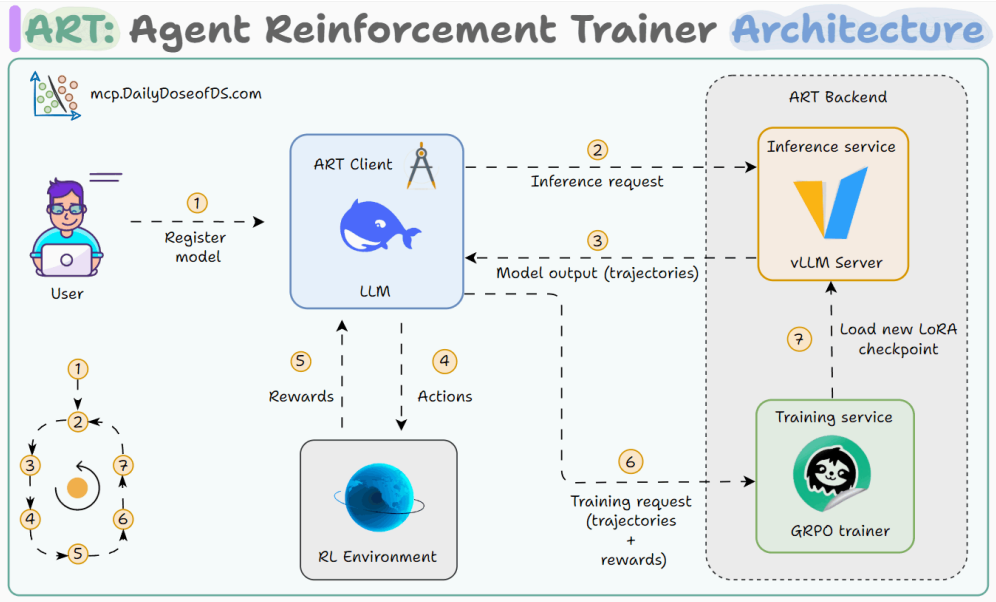
Agent Reinforcement Trainer(ART)

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:

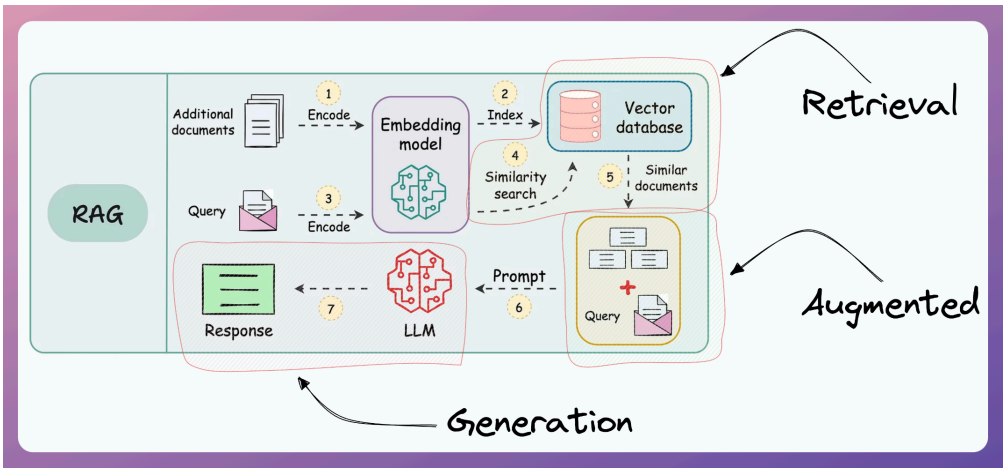
- Prompt Engineering - which steers the model at inference time
- Fine-tuning - which adjusts its internal parameters.

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.

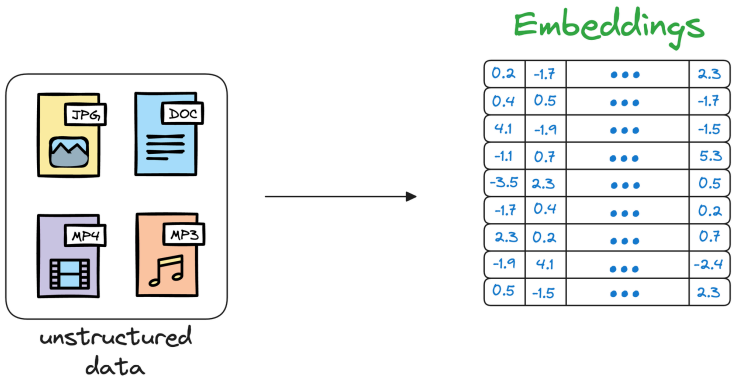
With RAG, the language model can use the retrieved information (which is expected to be reliable) from the vector database to ensure that its responses are grounded in real-world knowledge and context, reducing the likelihood of hallucinations.

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?

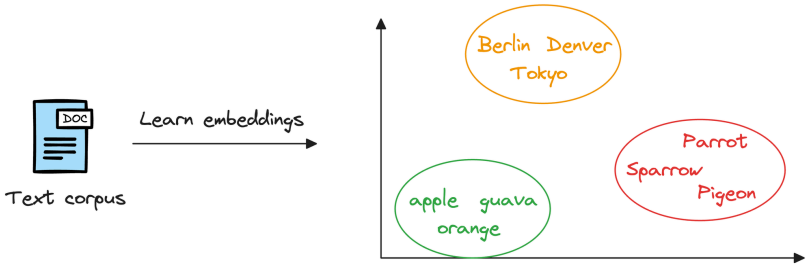
Simply put, a vector database stores unstructured data (text, images, audio, video, etc.) in the form of vector embeddings.



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).

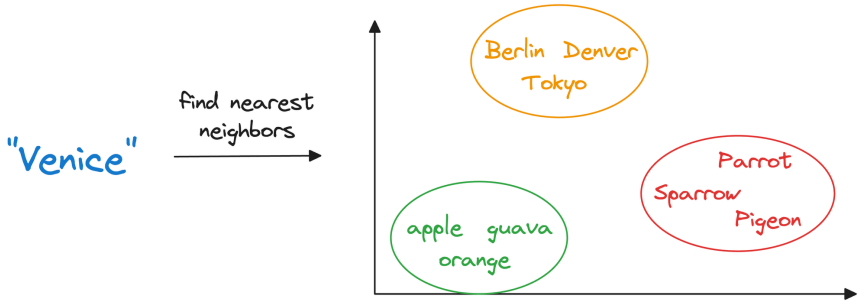
This numerical vector is called an embedding, and the model is trained in such a way that these vectors capture the essential features and characteristics of the underlying data.

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

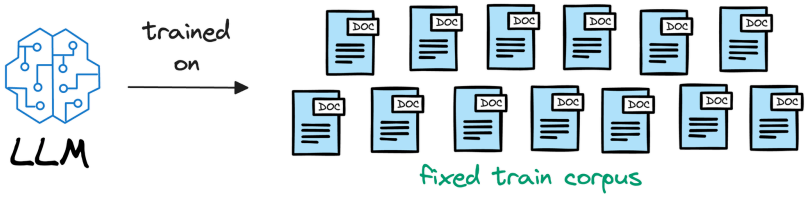
At this point, one interesting thing to learn is how exactly LLMs take advantage of vector databases.

The biggest confusion that people typically face is:

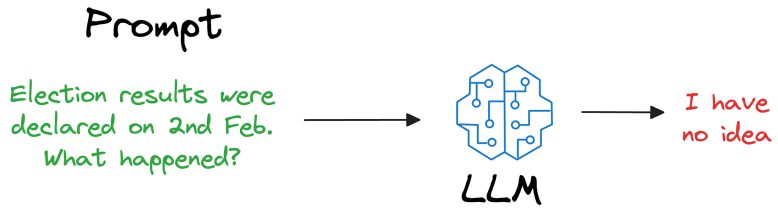
Once we have trained our LLM, it will have some model weights for text generation. Where do vector databases fit in here?

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.



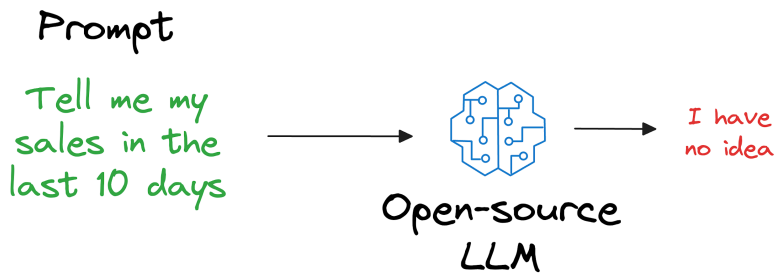
For instance, if the model was deployed after considering the data until 31st Jan 2024, and we use it, say, a week after training, it will have no clue about what happened in those days.



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.

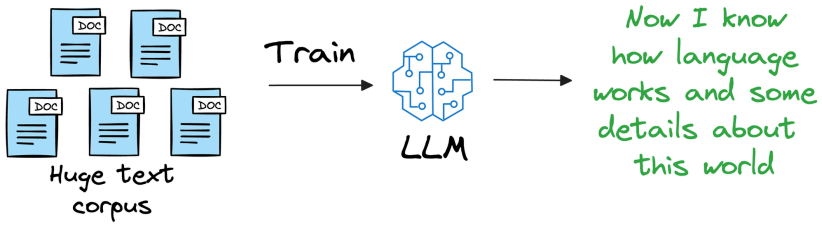


But if you think about it, is it really our objective to train an LLM to know every single thing in the world?

Not at all!

That's not our objective.

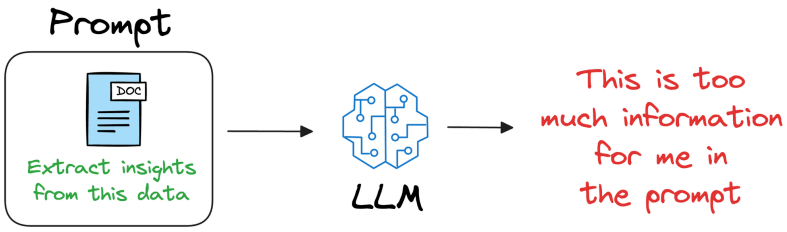
Instead, it is more about helping the LLM learn the overall structure of the language, and how to understand and generate it.



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.

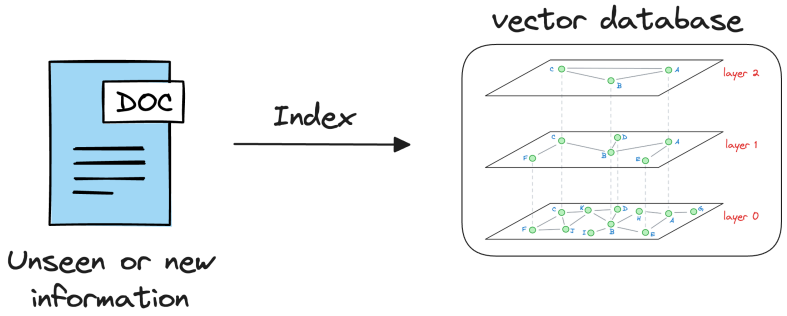


But since LLMs usually have a limit on the context window (number of words/tokens they can accept), the additional information can exceed that limit.

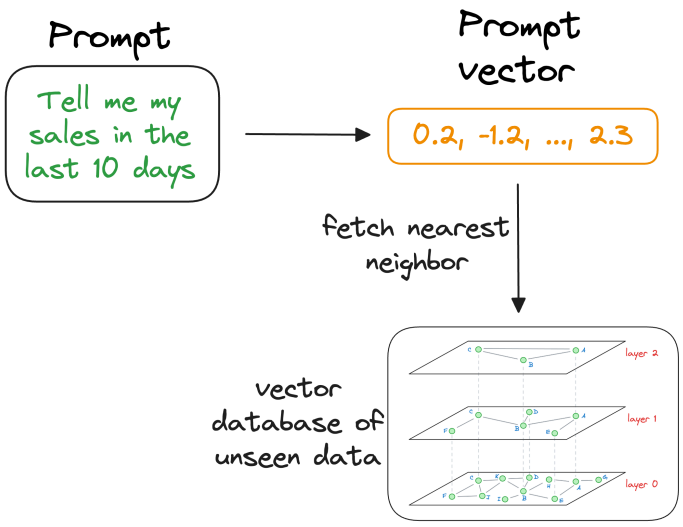
Vector databases solve this problem.

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.

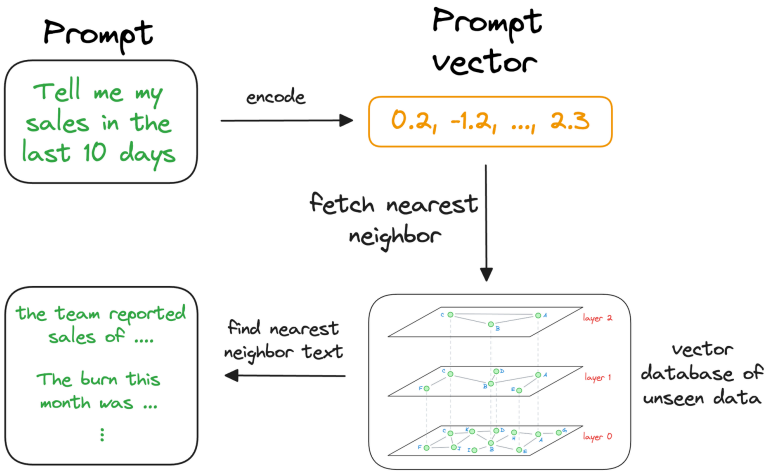
Thus, we can maintain our available information in a vector database by encoding it into vectors using an embedding model.



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.

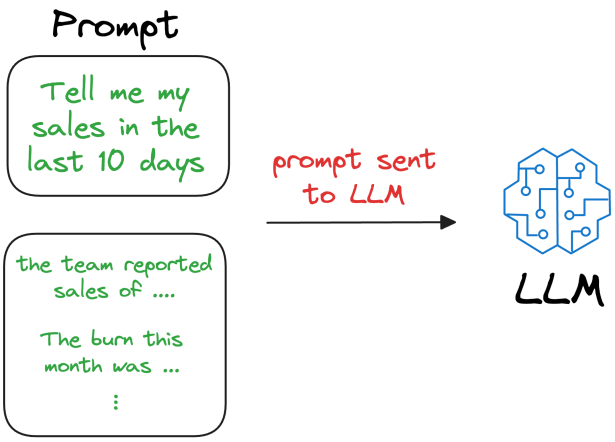


Once the approximate nearest neighbors have been retrieved, we gather the context corresponding to those specific vectors, which were stored at the time of indexing the data in the vector database (this raw data is stored as payload, which we will learn during implementation).



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.



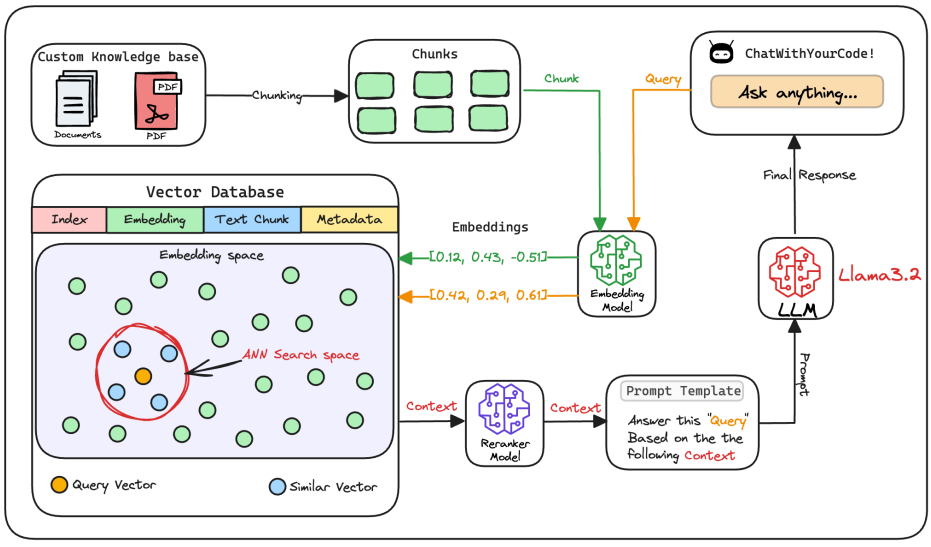
Consequently, the LLM can easily incorporate this info while generating text because it now has the relevant details available in the prompt.

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

Workflow of a RAG system

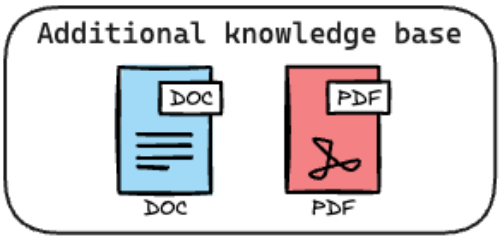
To build a RAG system, it's crucial to understand the foundational components that go into it and how they interact. Thus, in this section, let's explore each element in detail.

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



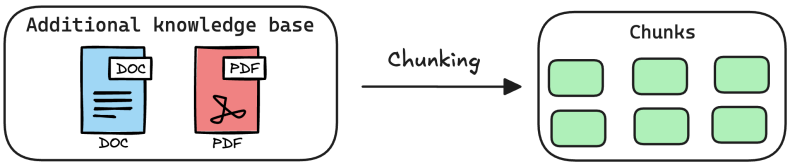
Let's break it down step by step.

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

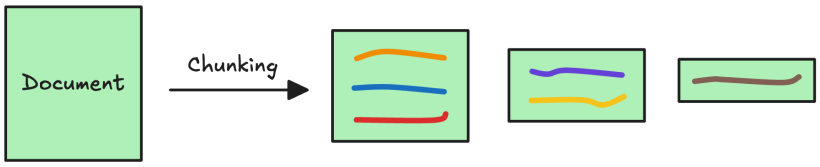


#1) Create chunks

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



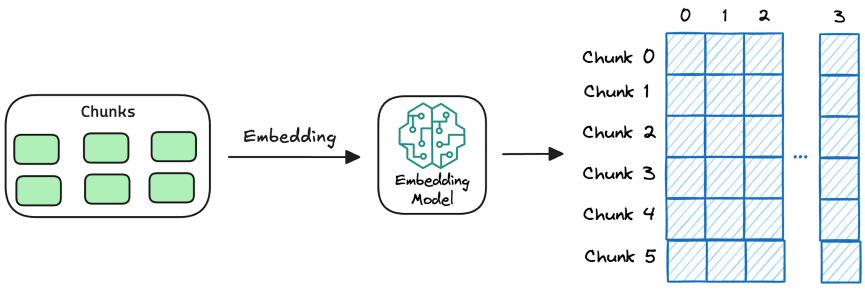
We do this because the additional document(s) can be pretty large. Thus, it is important to ensure that the text fits the input size of the embedding model.



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

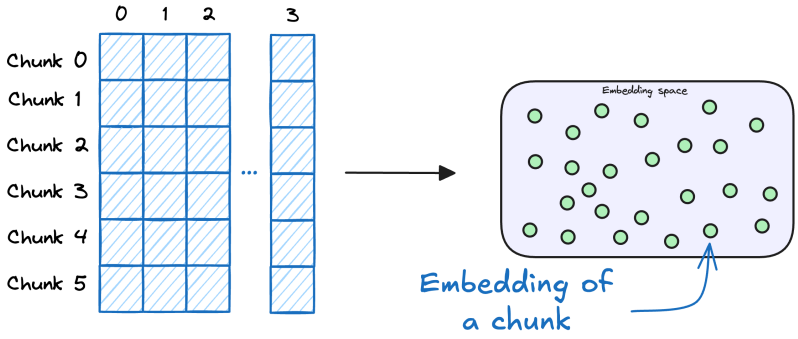
After chunking, we embed the chunks using an embedding model.



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.

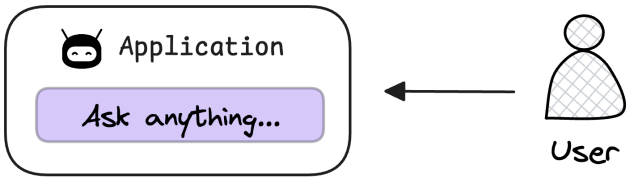
A vector database also stores the metadata and original content along with the vector embeddings.

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.

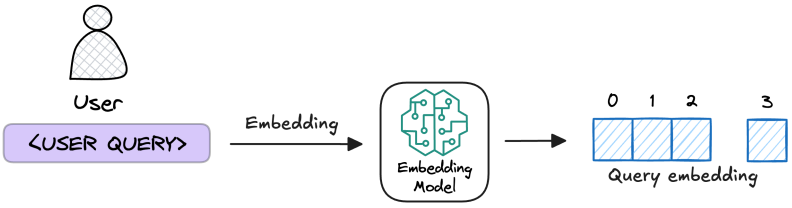
#4) User input query

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



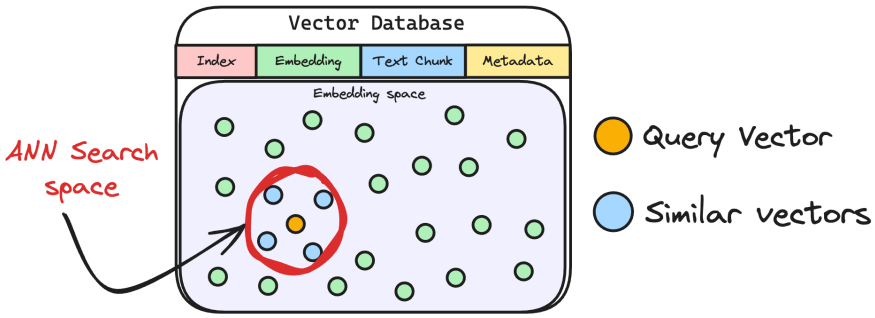
#5) Embed the query

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

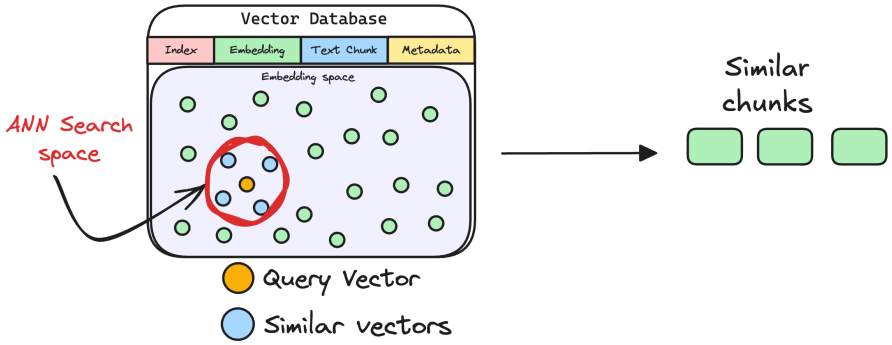


#6) Retrieve similar chunks

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).

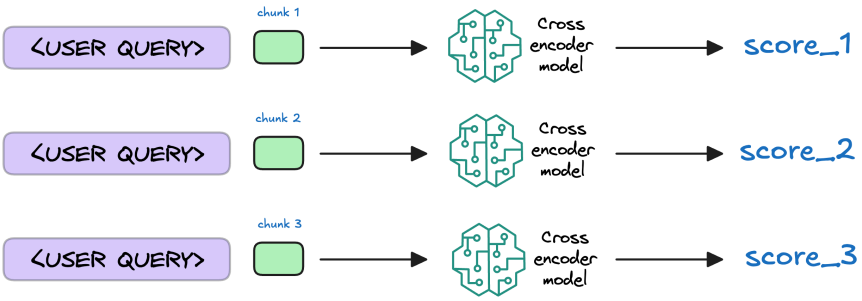


It is expected that these retrieved documents contain information related to the query, providing a basis for the final response generation.

#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.

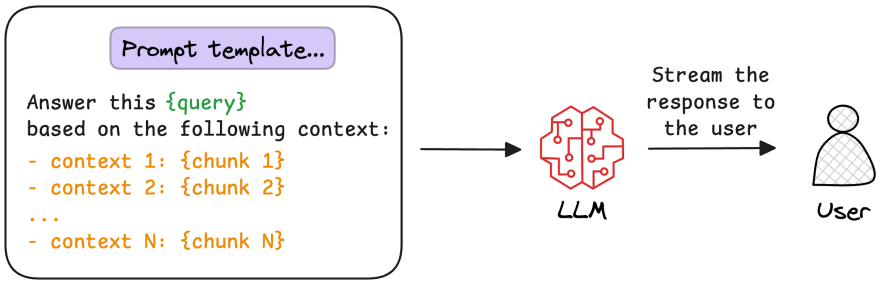
#8) Generate the final response

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:

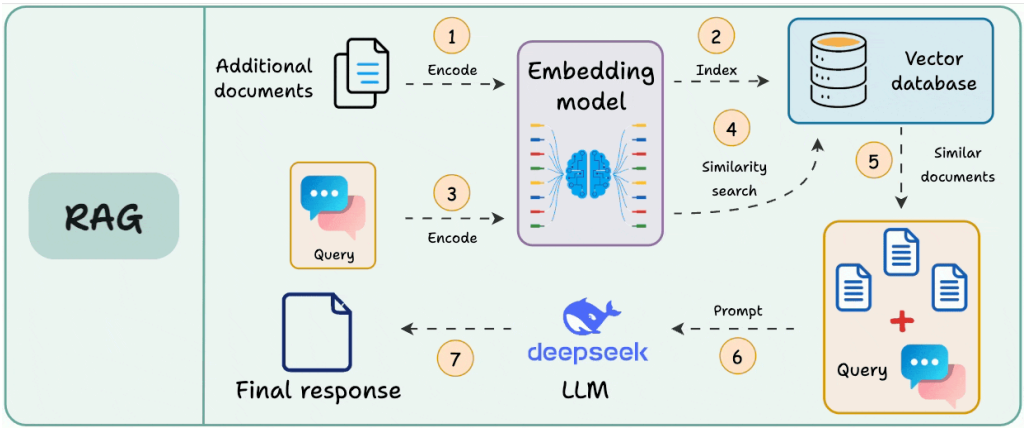


The LLM leverages the context provided by the chunks to generate a coherent and contextually relevant answer that directly addresses the user's query.

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

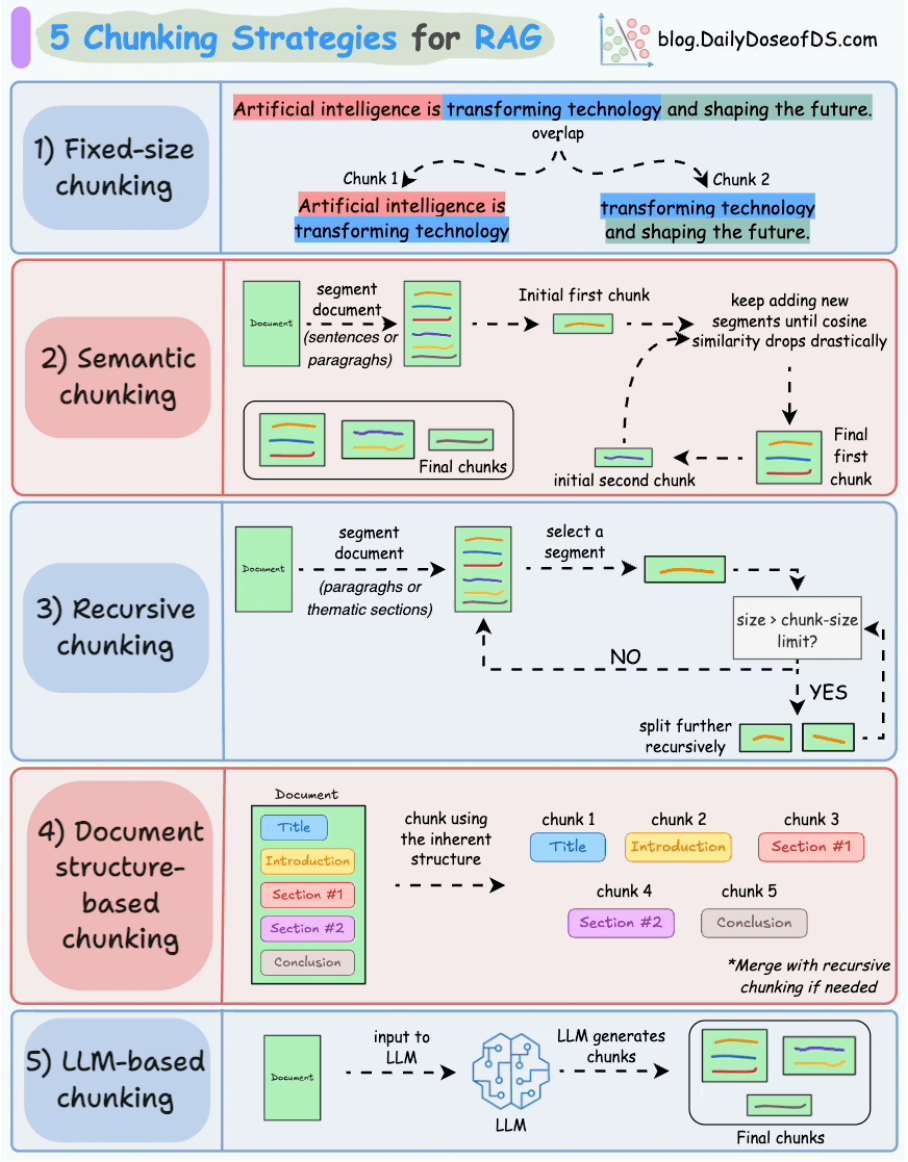
Here's the typical workflow of RAG:



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.

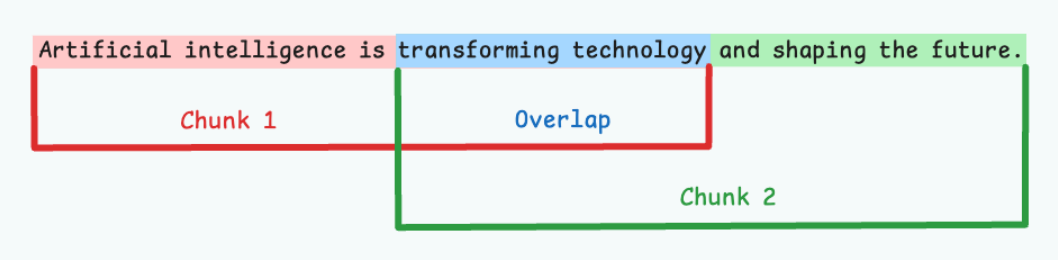
Here are five chunking strategies for RAG:



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.

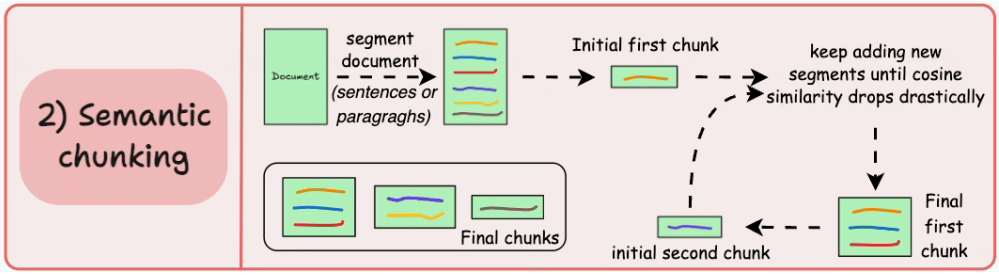


Since a direct split can disrupt the semantic flow, it is recommended to maintain some overlap between two consecutive chunks (the blue part above).

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:

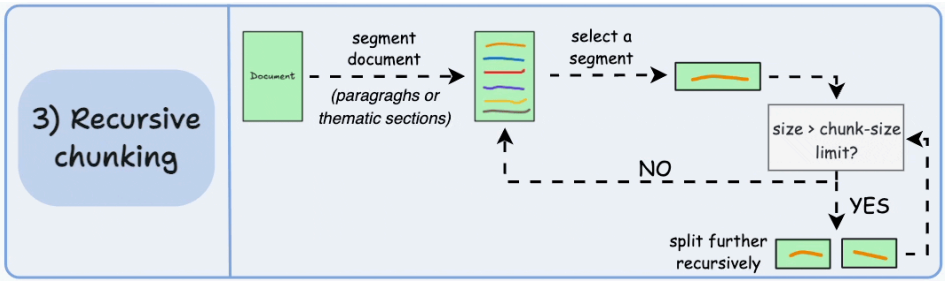
```
Artificial intelligence is transforming industries by automating
processes, enhancing decision-making, and providing insights through
data analysis. Machine learning, a subset of AI, enables systems to learn
and improve from experience without explicit programming. Deep learning,
a branch of machine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

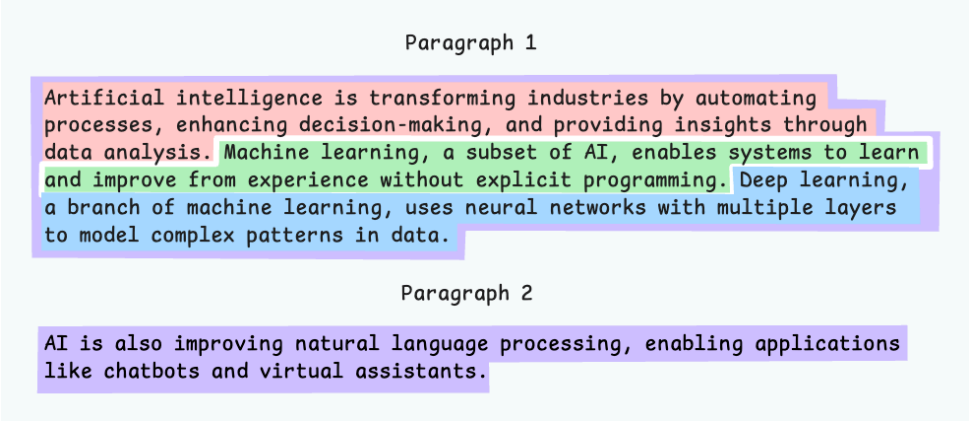
3) Recursive chunking



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:

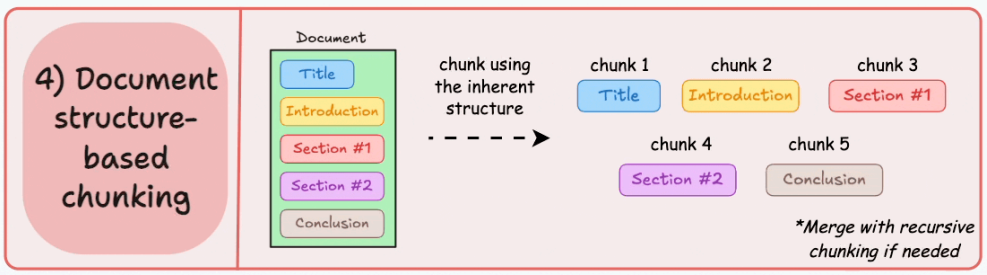
First, we define two chunks (the two paragraphs in purple).

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.

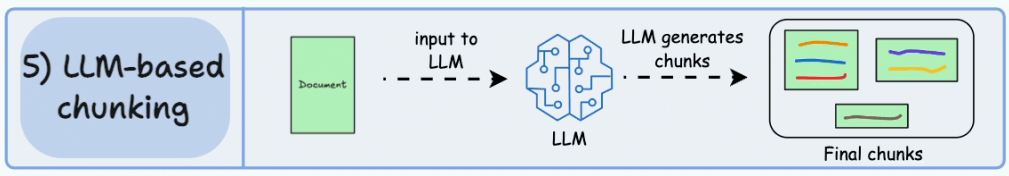
Here's what the output could look like:

Title: The Role of Artificial Intelligence in Modern Education	Chunk 1
Introduction Artificial intelligence (AI) is reshaping education by providing personalized learning experiences and automating administrative tasks.	Chunk 2
Section 1: Personalized Learning AI enables the customization of educational content to meet individual student needs, enhancing engagement and comprehension.	Chunk 3
Section 2: Administrative Automation From grading to scheduling, AI tools are streamlining administrative processes, allowing educators to focus more on teaching.	Chunk 4
Conclusion The integration of AI in education holds the promise of more efficient learning environments and improved student outcomes.	Chunk 5

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.

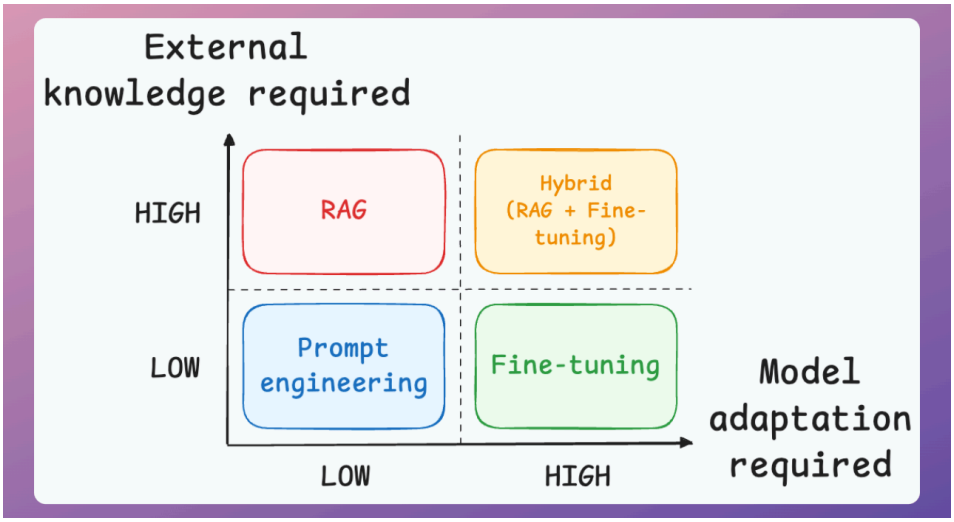
The choice will depend on the nature of your content, the capabilities of the embedding model, computational resources, etc.

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:

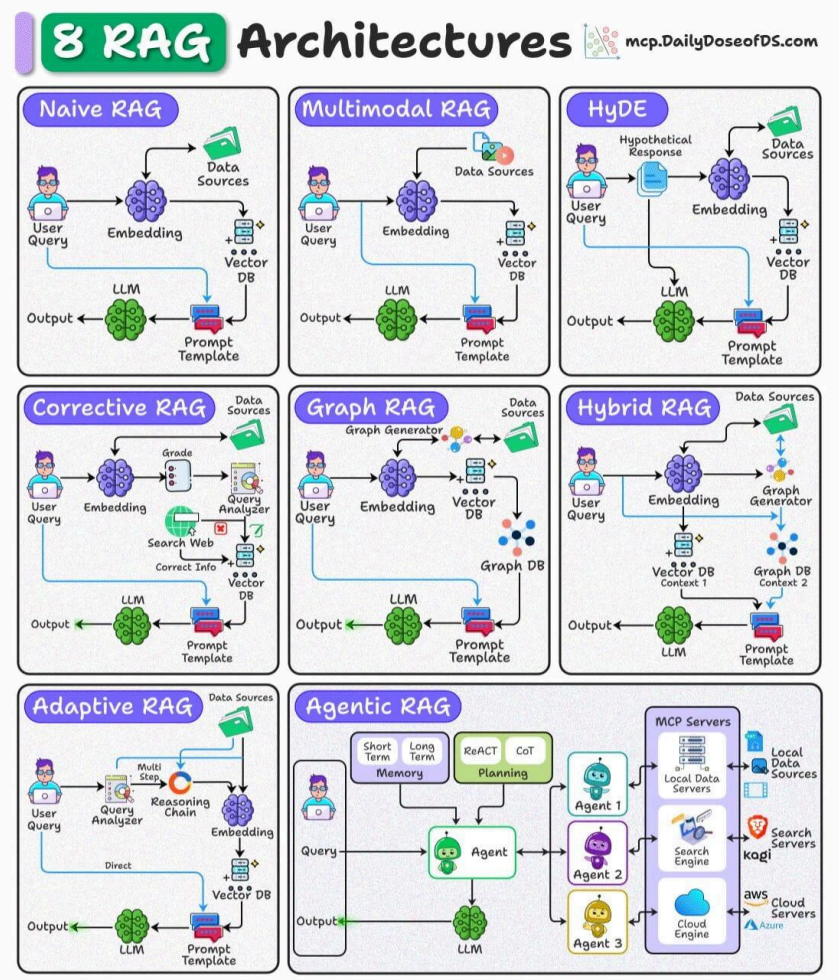
- Use RAGs to generate outputs based on a custom knowledge base if the vocabulary & writing style of the LLM remains the same.
- Use fine-tuning to change the structure (behaviour) of the model than knowledge.
- Prompt engineering is sufficient if you don't have a custom knowledge base and don't want 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.

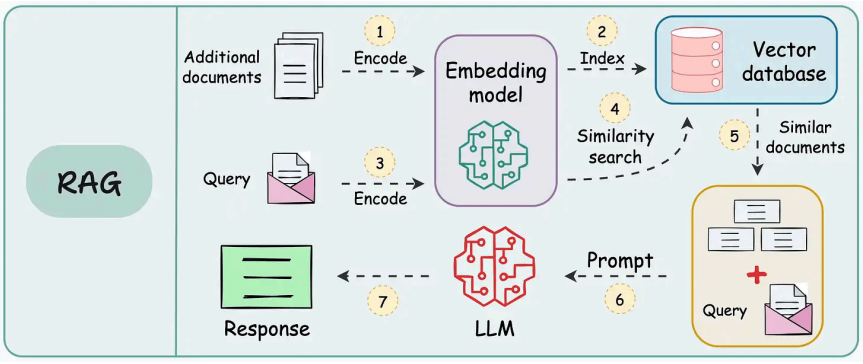
8) Agentic RAG

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.

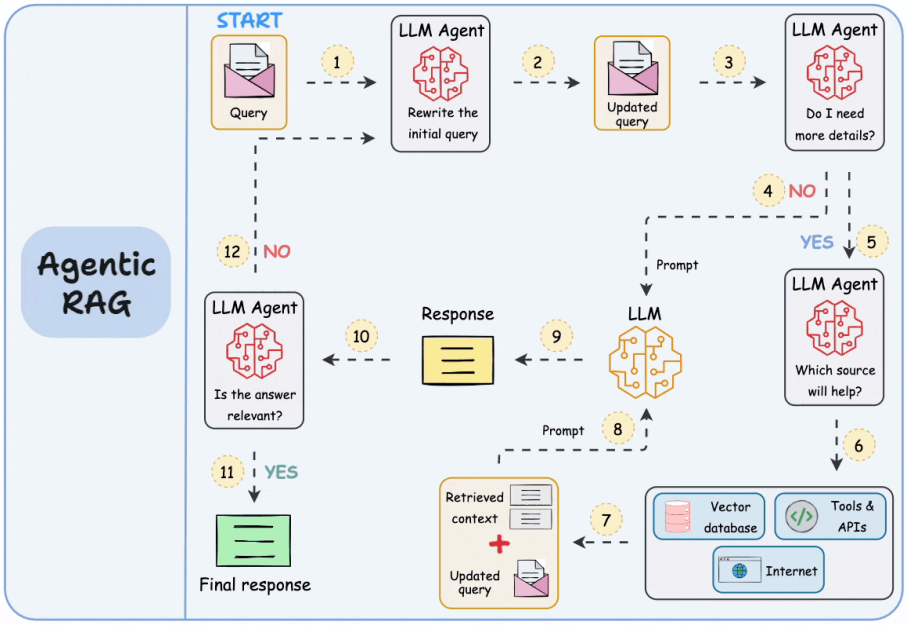
RAG systems may provide relevant context but don't reason through complex queries. If a query requires multiple retrieval steps, traditional RAG falls short.

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.

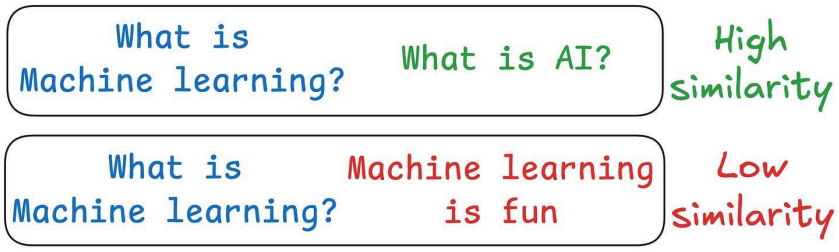
This makes the RAG much more robust since, at every step, agentic behavior ensures that individual outcomes are aligned with the final goal.

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

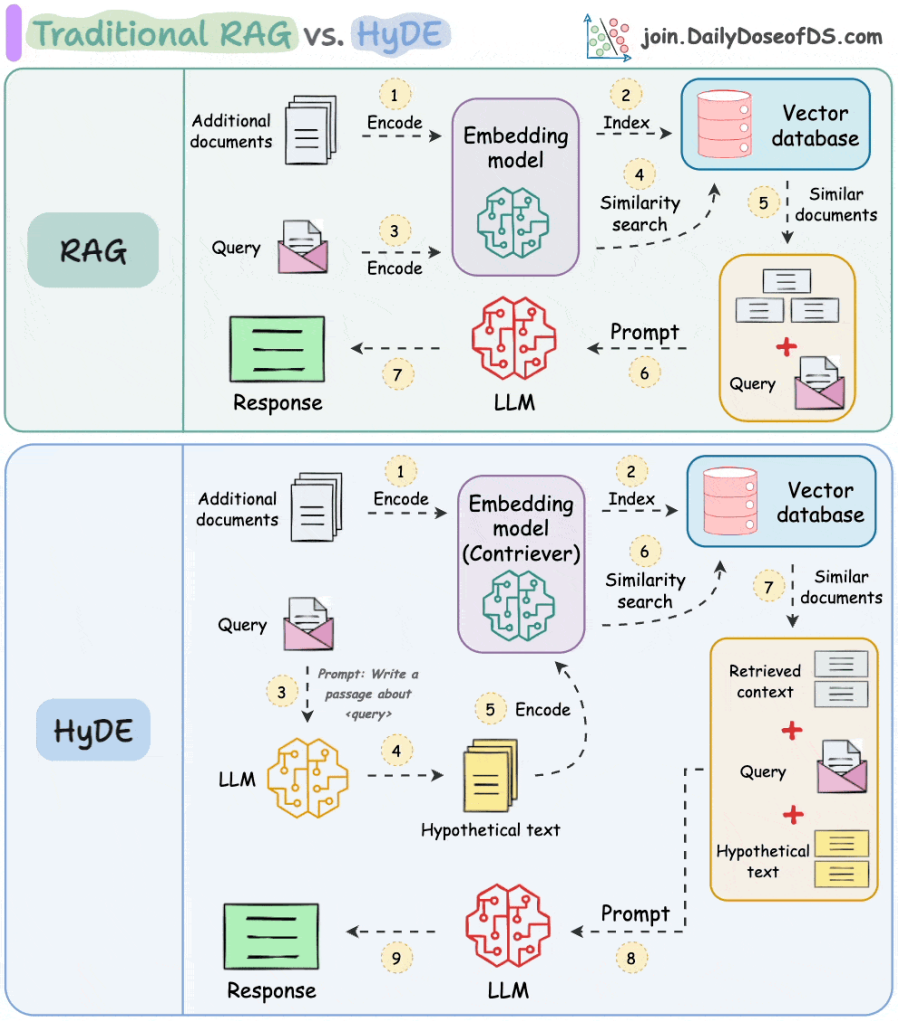
Another critical problem with the traditional RAG system is that questions are not semantically similar to their answers.



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.

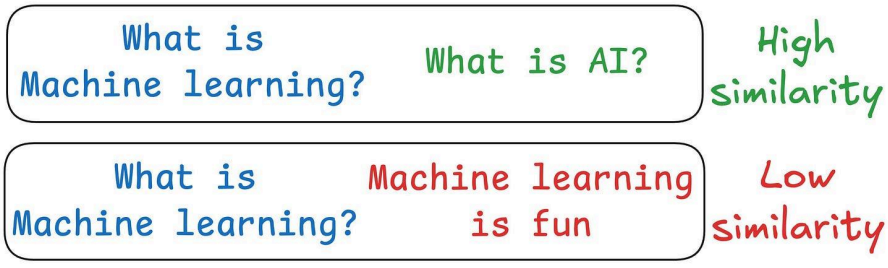
HyDE solves this.

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:

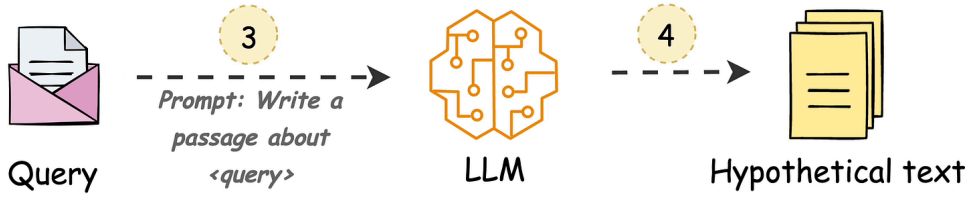
Use an LLM to generate a hypothetical answer H for the query Q (this answer does not have to be entirely correct).

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.

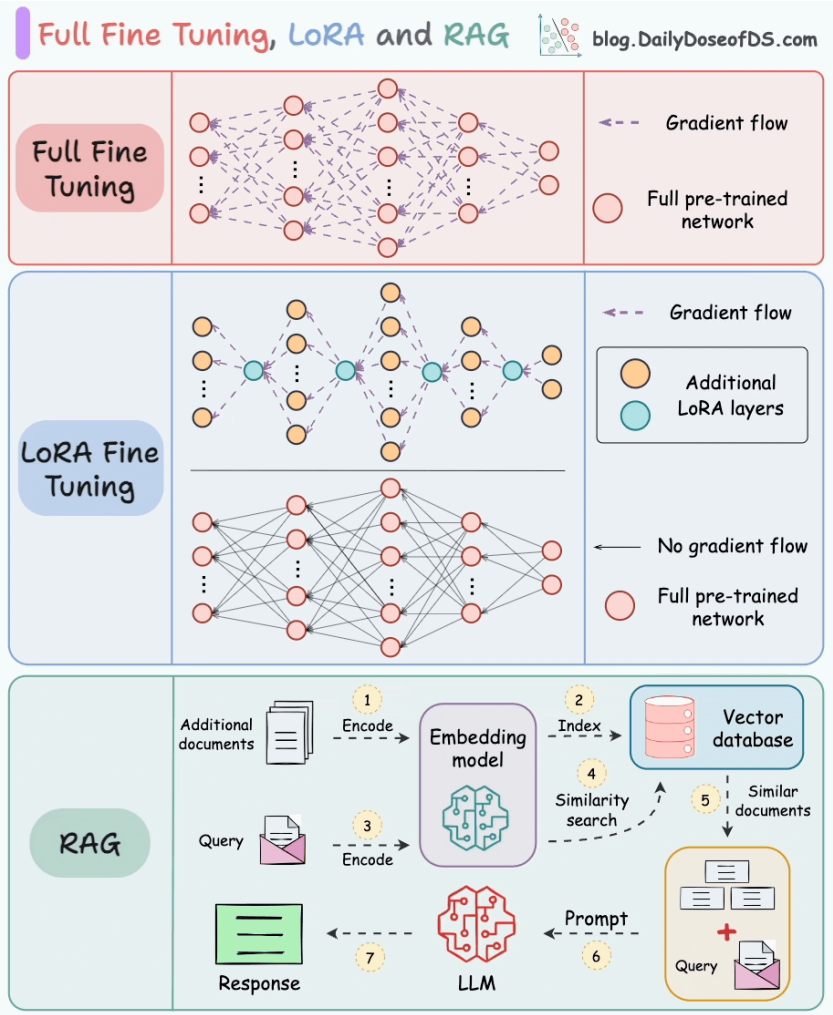
More specifically, this model is trained using contrastive learning and it also functions as a near-lossless compressor whose task is to filter out the hallucinated details of the fake document.

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.

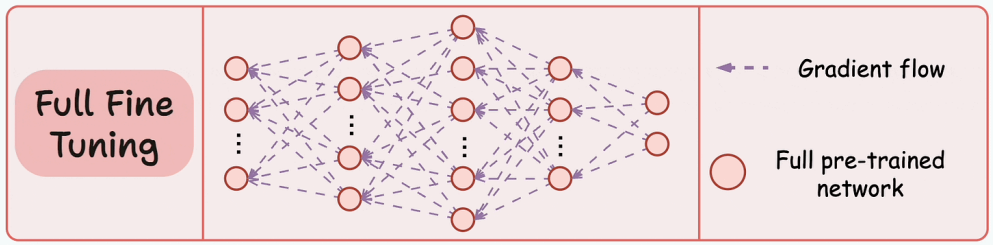
Full-model Fine-tuning vs. LoRA vs. RAG



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.

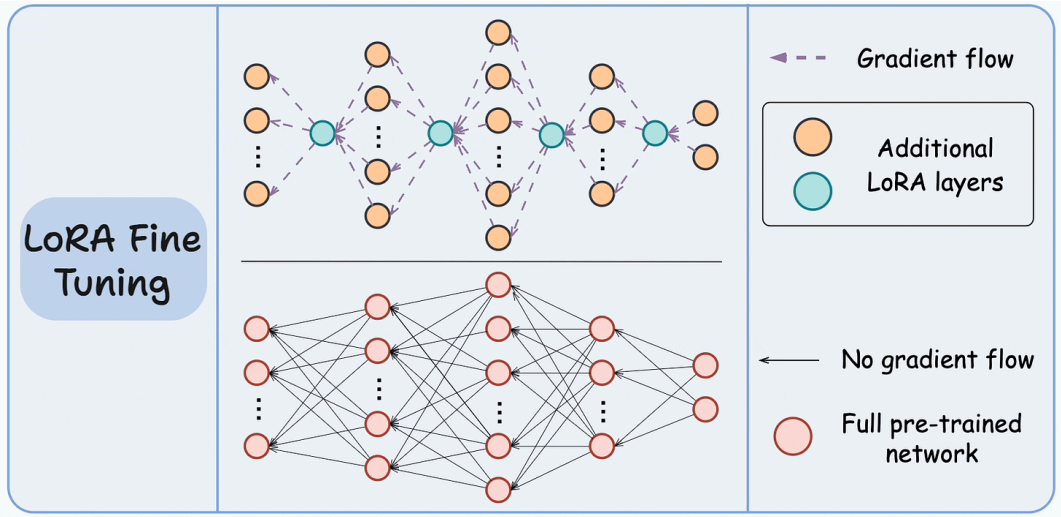
The cost involved in maintaining all large fine-tuned models.

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.

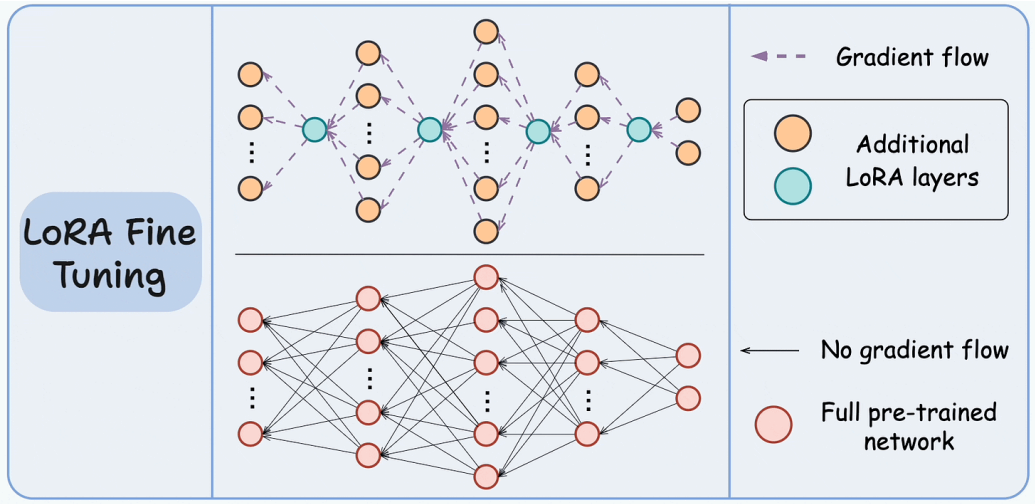
Looking at the above visual, you might think:

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:

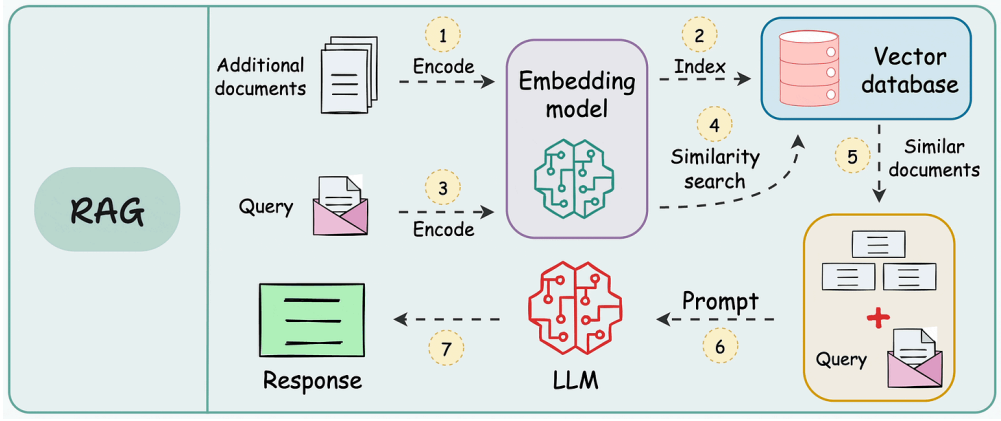


Looking at the above visual, it is pretty clear that the LoRA network has relatively very few connections.

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

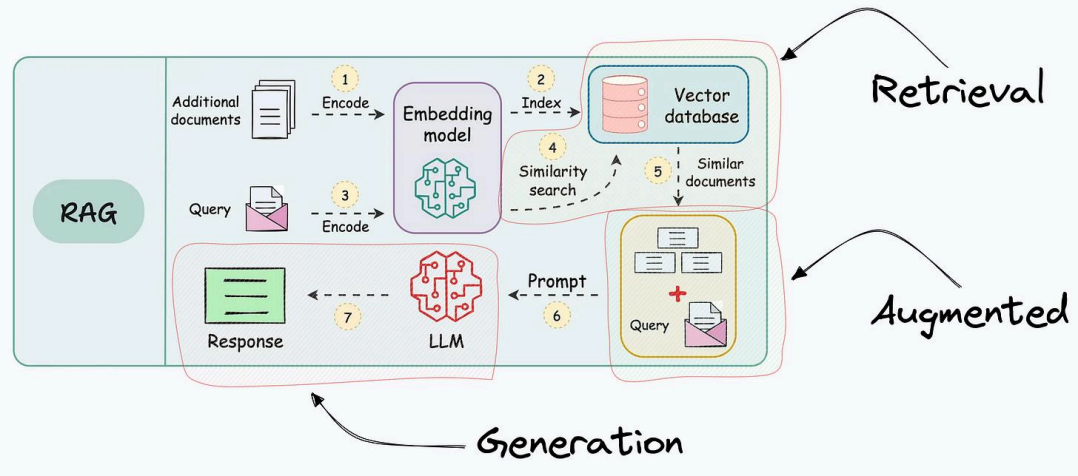
embeddings into the vector database. There's no need to repeat this again for the entire data)

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:

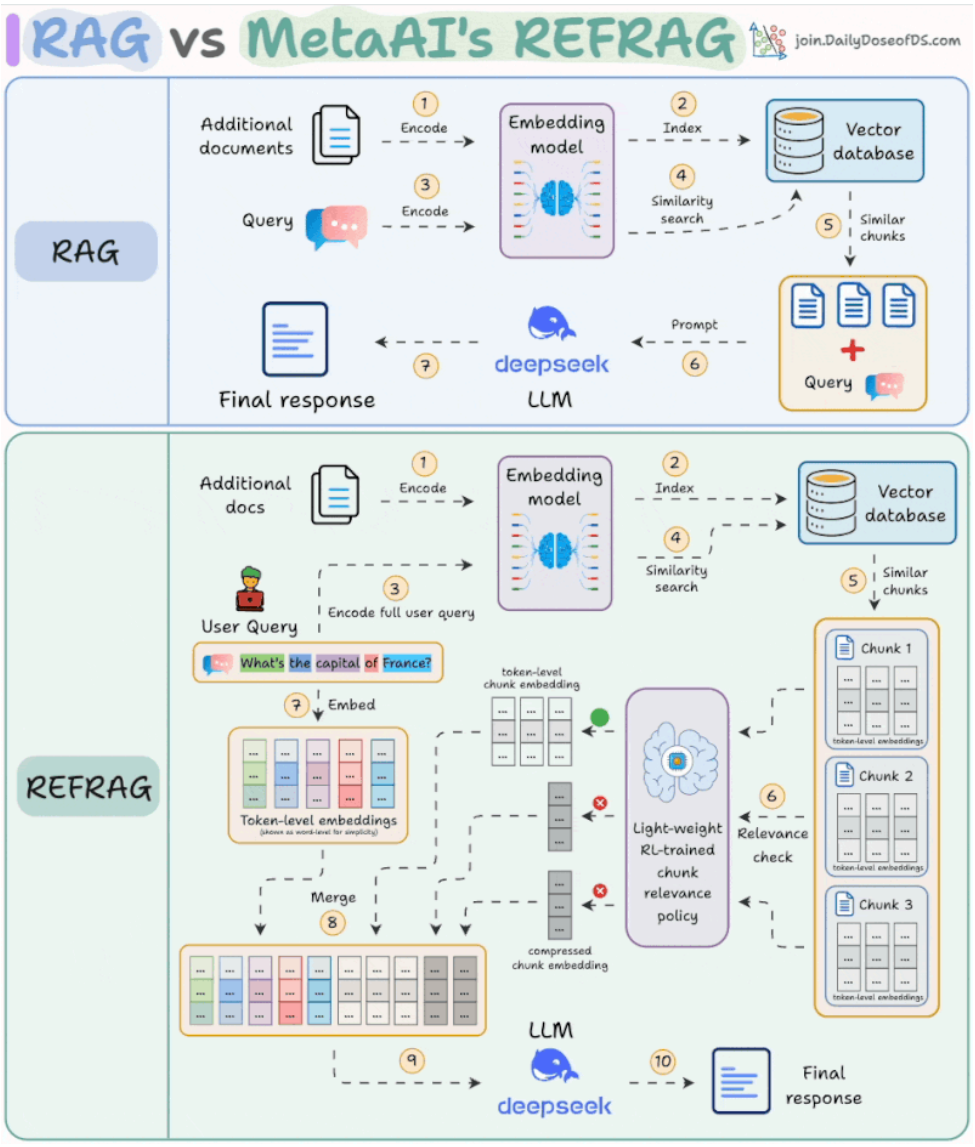
- You encode it into a vector.
- Fetch similar chunks from vector DB.
- Dump the retrieved context into the LLM.

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

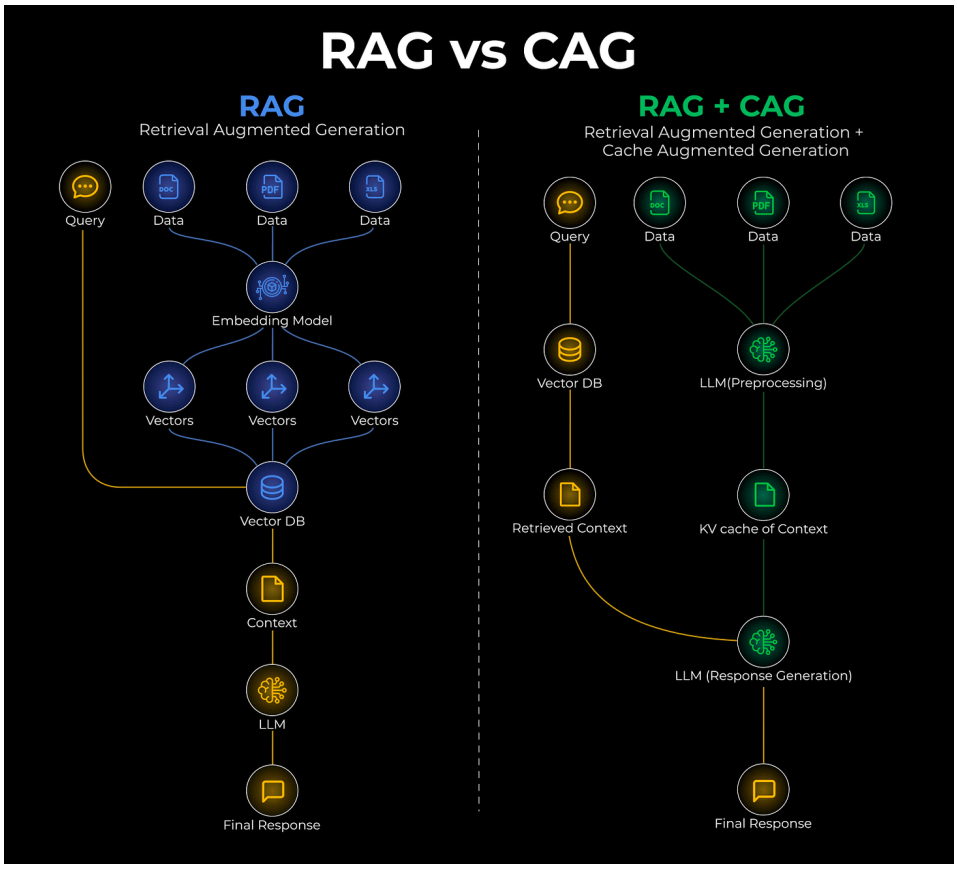
RAG changed how we build knowledge-grounded systems, but it still has a weakness.

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.

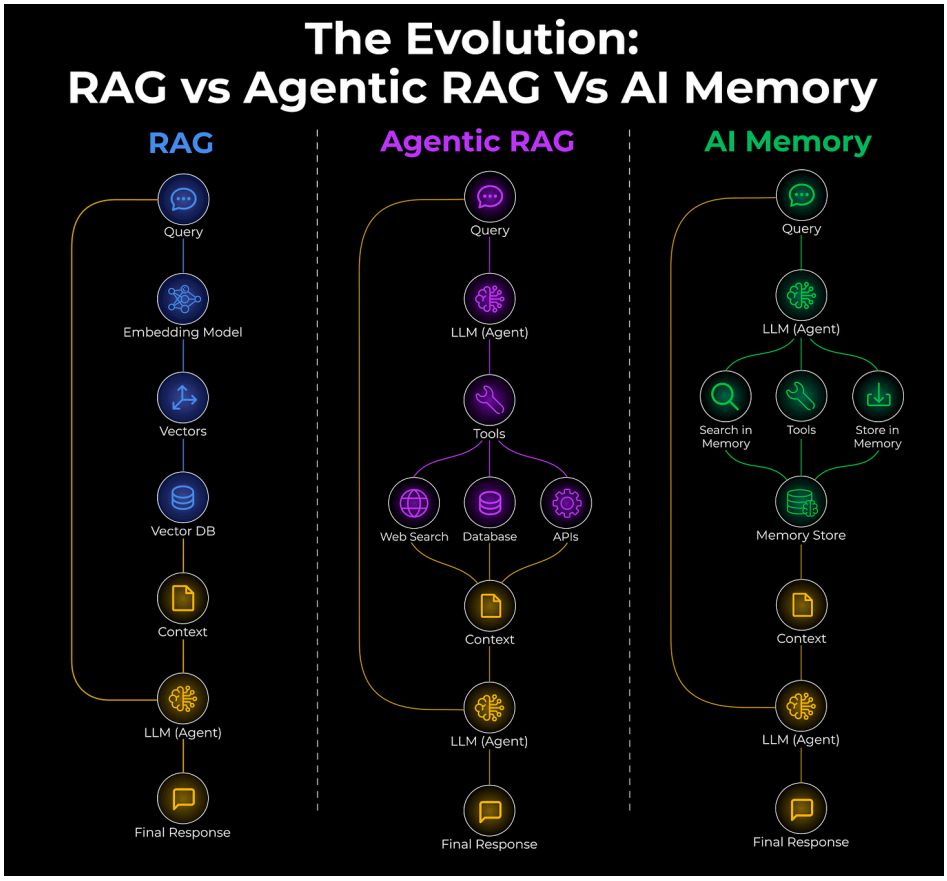
You can also see this in practice above.

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

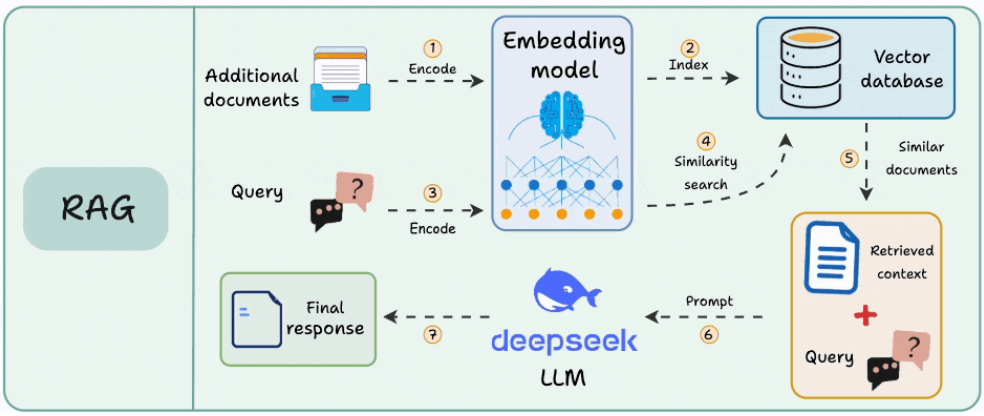
What is Context Engineering?

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.

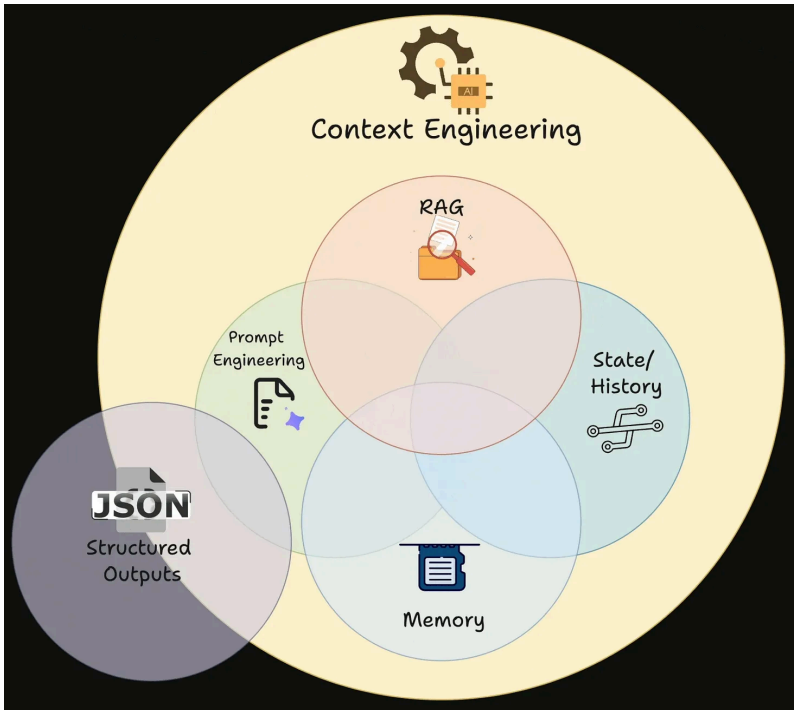


Thus:

- 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.

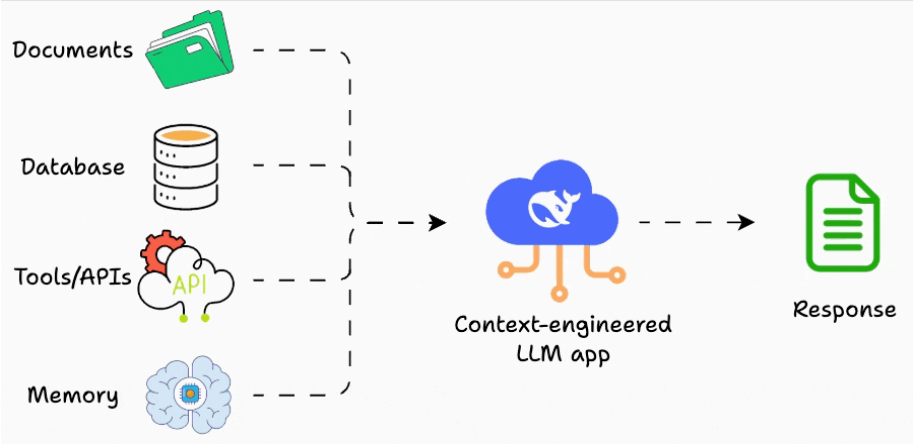
But why was traditional prompt engineering not enough?

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…

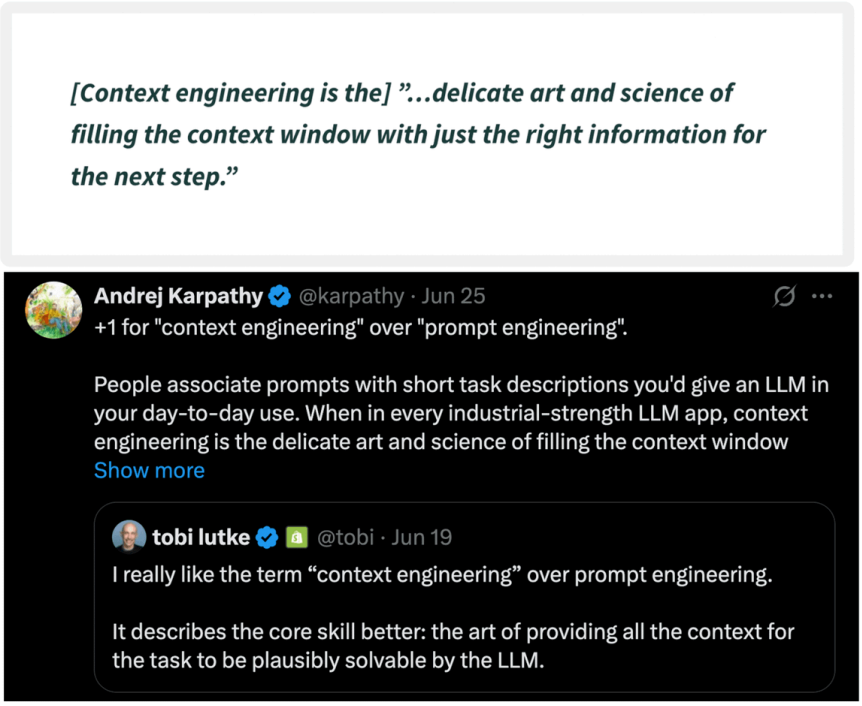
Context engineering is becoming the new core skill since it addresses the real bottleneck, which is not model capability, but setting up an architecture of information.

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.

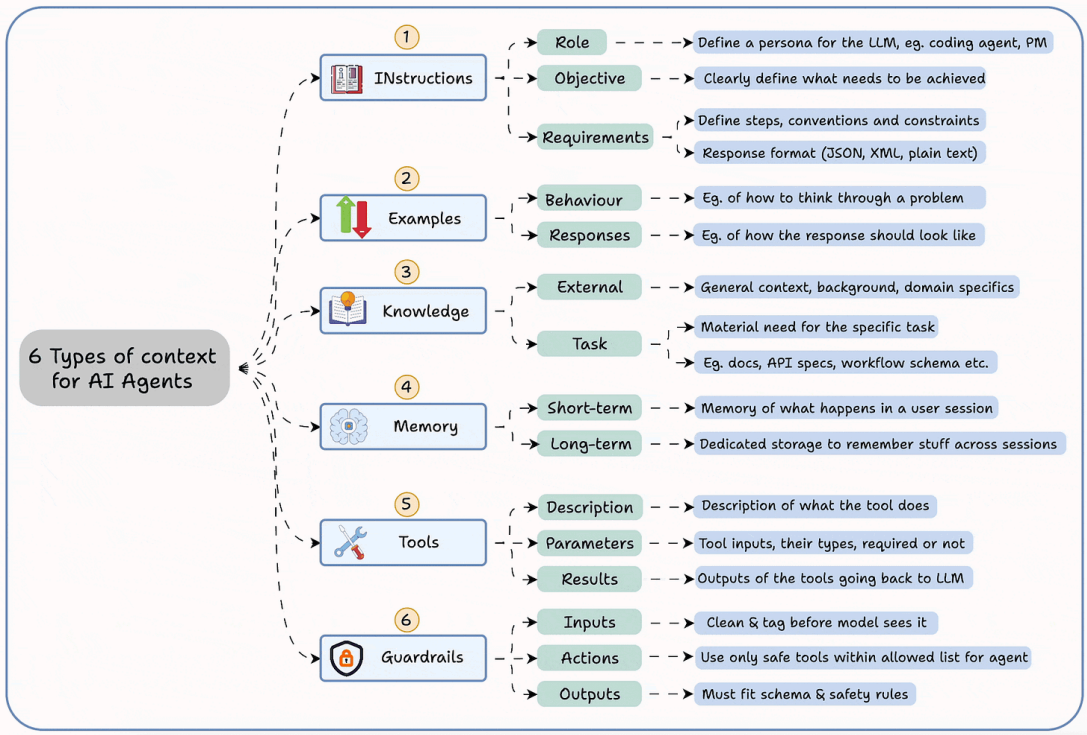
Here's a quote by Andrej Karpathy on context engineering...



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:

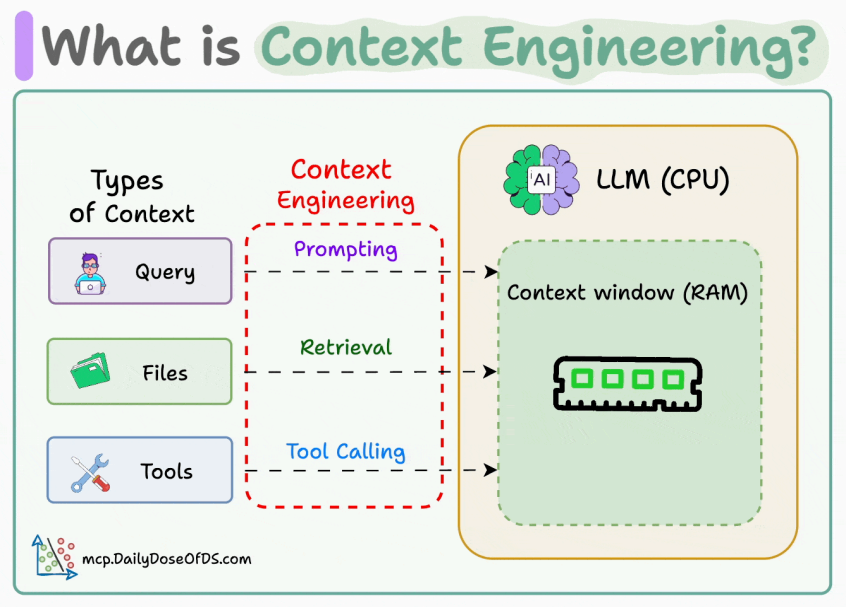


- Instructions
- Examples
- Knowledge
- Memory
- Tools
- Guardrails

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

You must engineer the input (context).

Think of it this way:

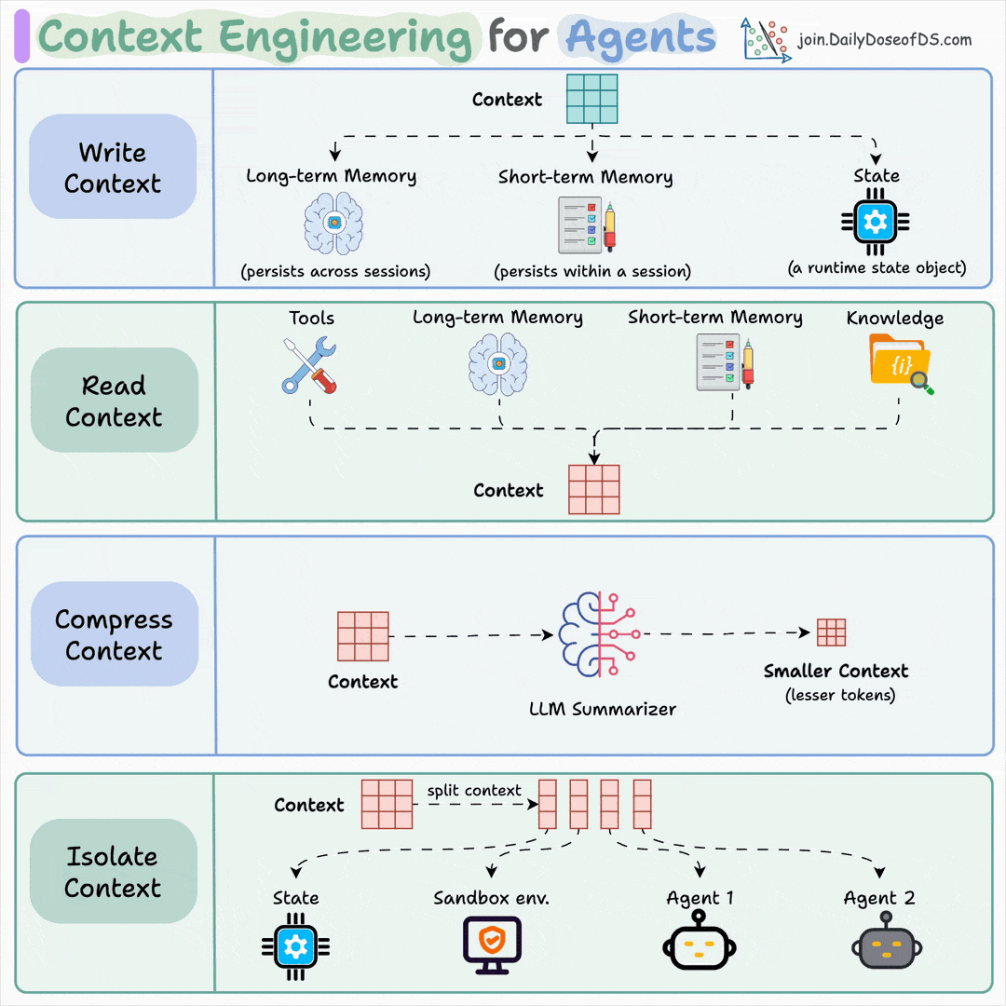


- If LLM is a CPU.
- Then the context window is the RAM.

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:

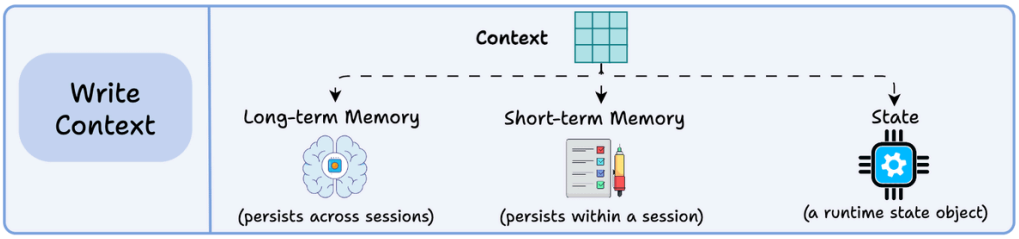


- Writing Context
- Selecting Context
- Compressing Context
- Isolating Context

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.

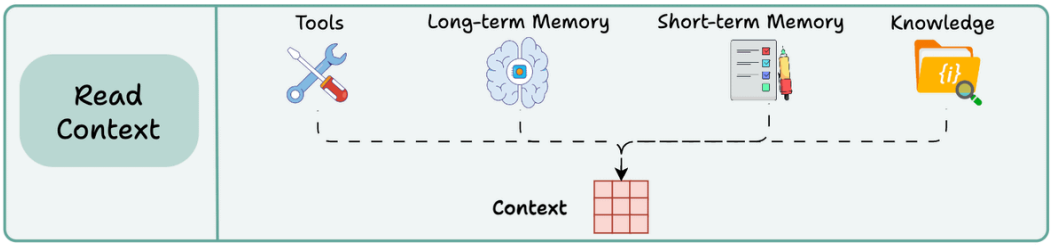


You can do so by writing it to:

- 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.

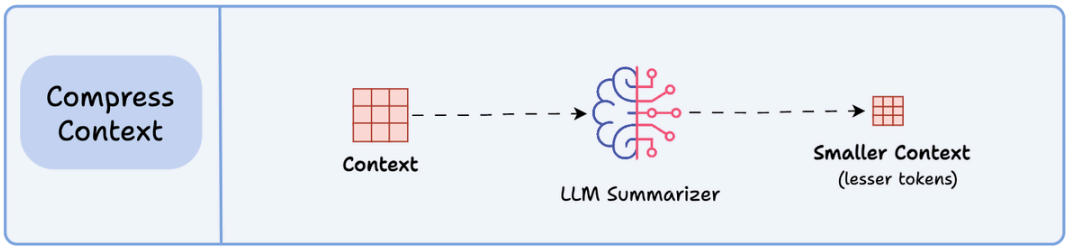


Now this context can be pulled from:

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

3) Compressing context

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

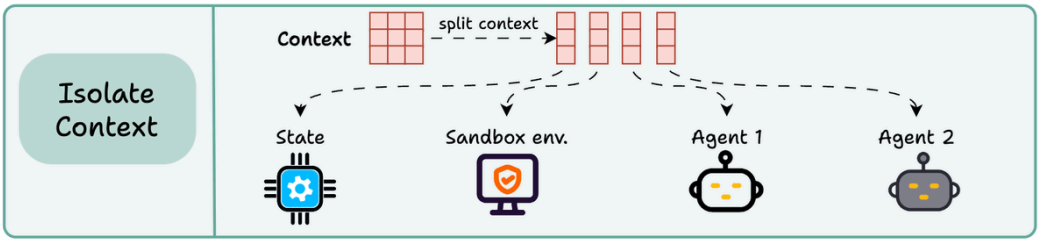


The retrieved context may contain duplicate or redundant information (multi-turn tool calls), leading to extra tokens & increased cost.

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!

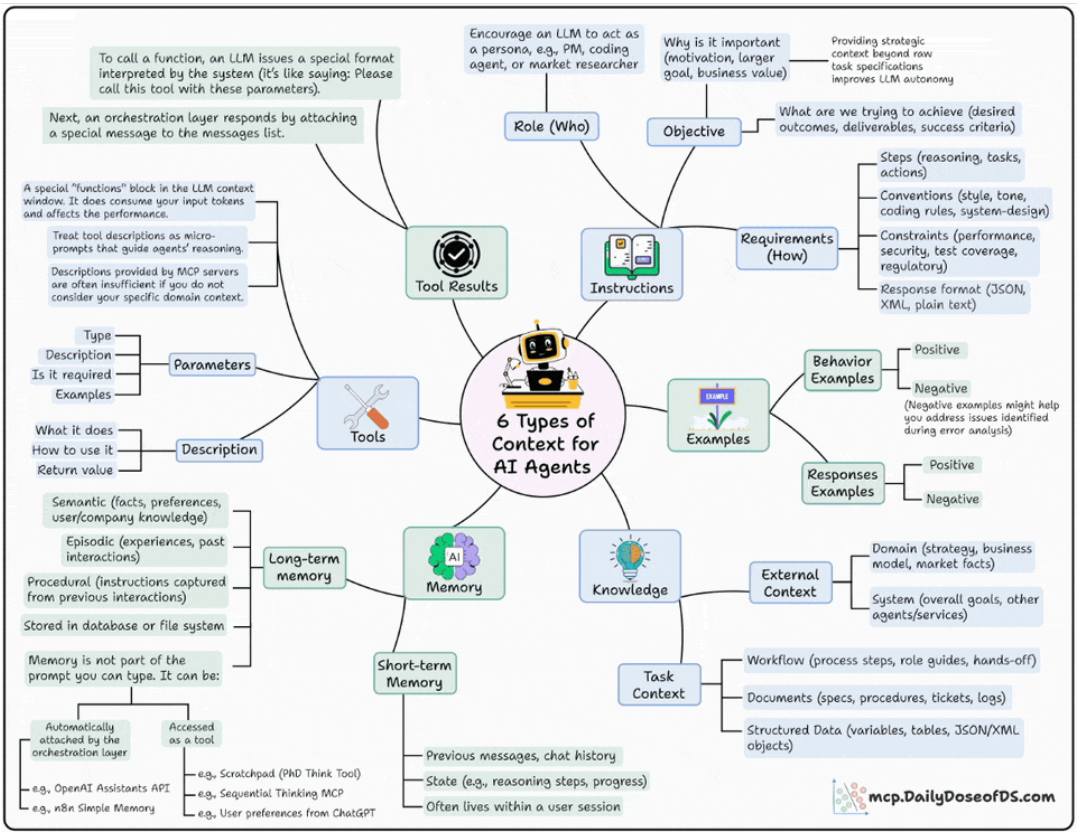
6 Types of Contexts for AI Agents

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.

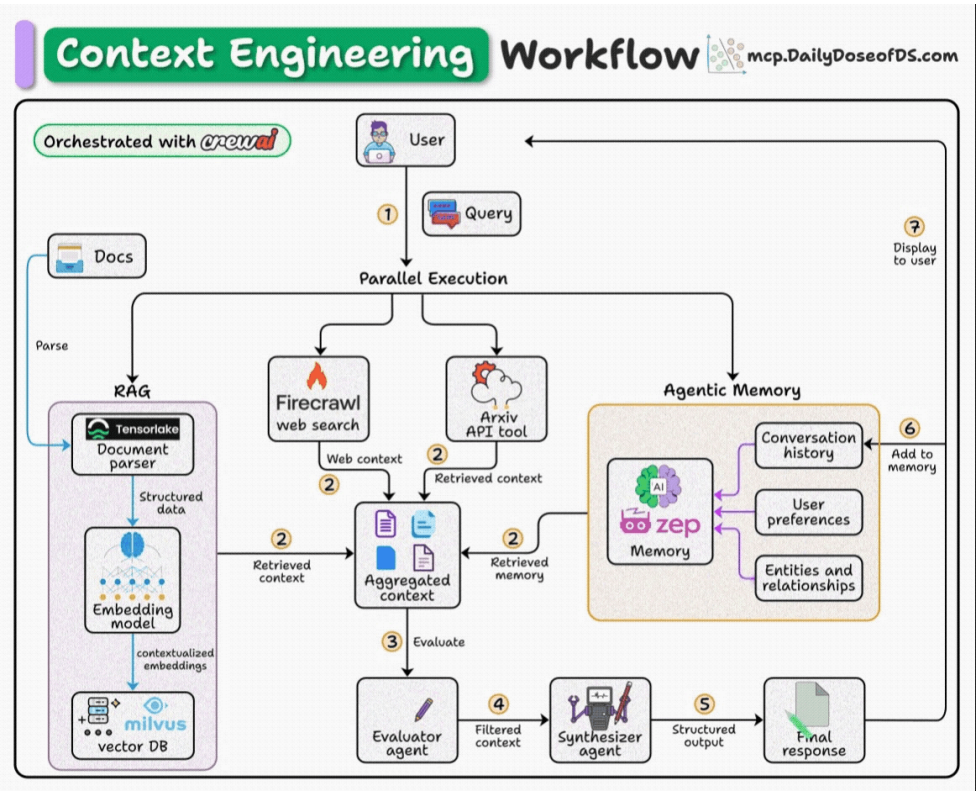
These are the exact six layers that help you build fully context-aware Agents.

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:

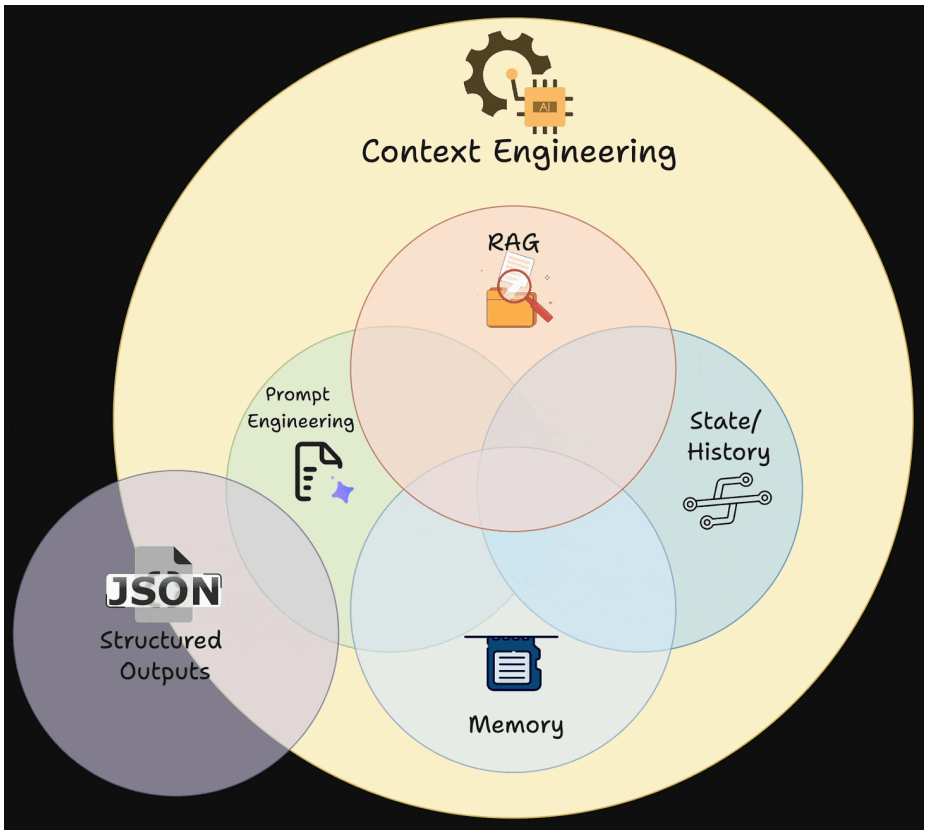


- User submits query.
- Fetch context from docs, web, arxiv API, and memory.
- Pass the aggregated context to an agent for filtering.
- Pass the filtered context to another agent to generate a response.
- Save the final response to memory.

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

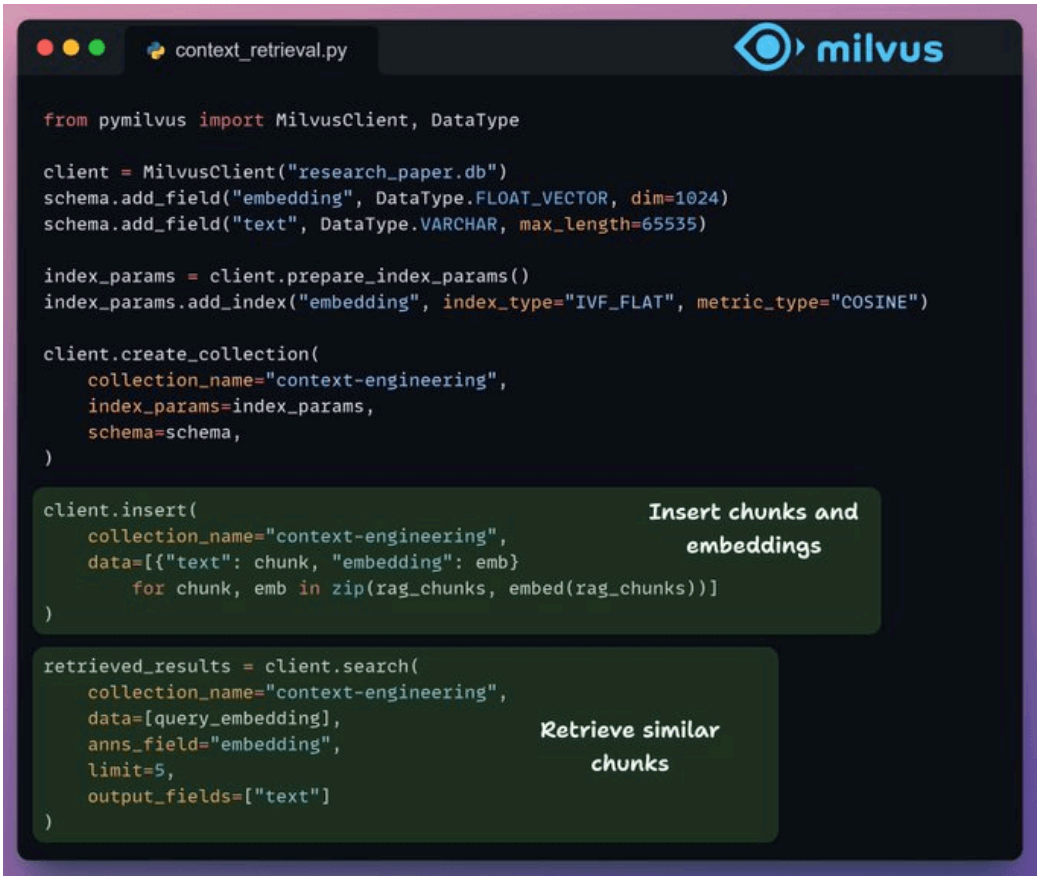
We use Tensorlake to convert the document into RAG-ready markdown chunks for each section.

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.

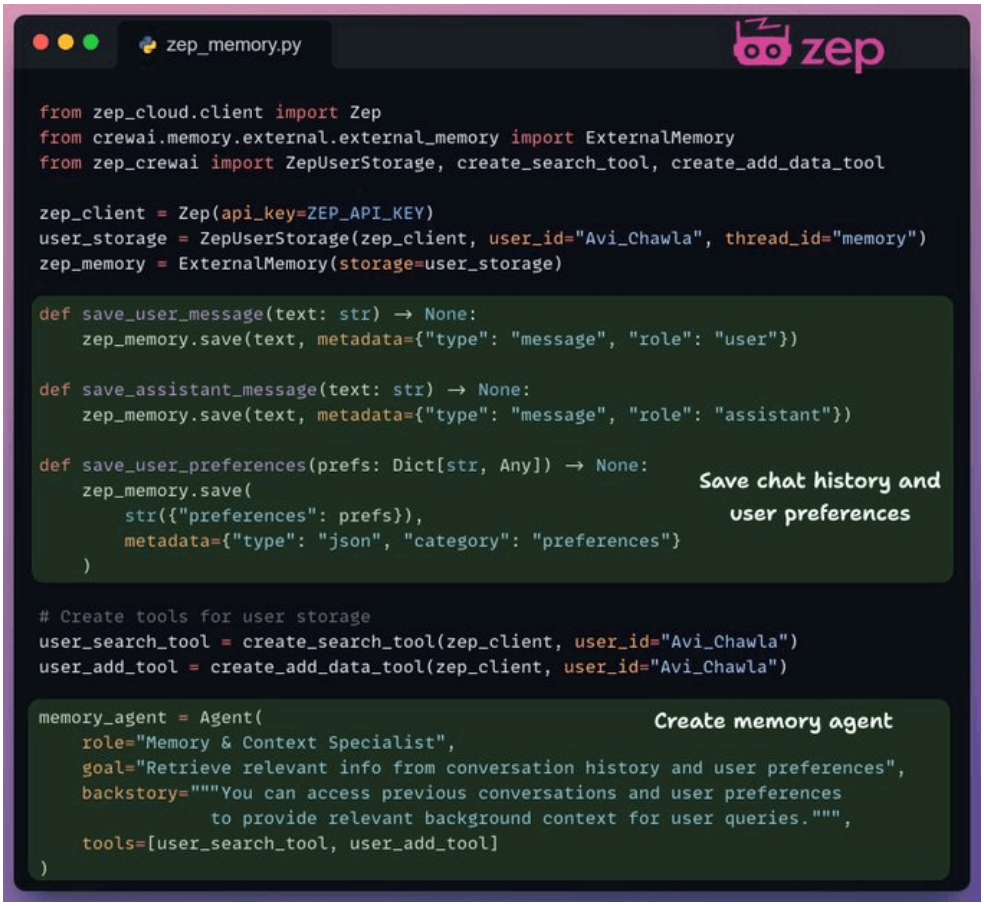
We retrieve the top-k most similar chunks to our query:



#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.

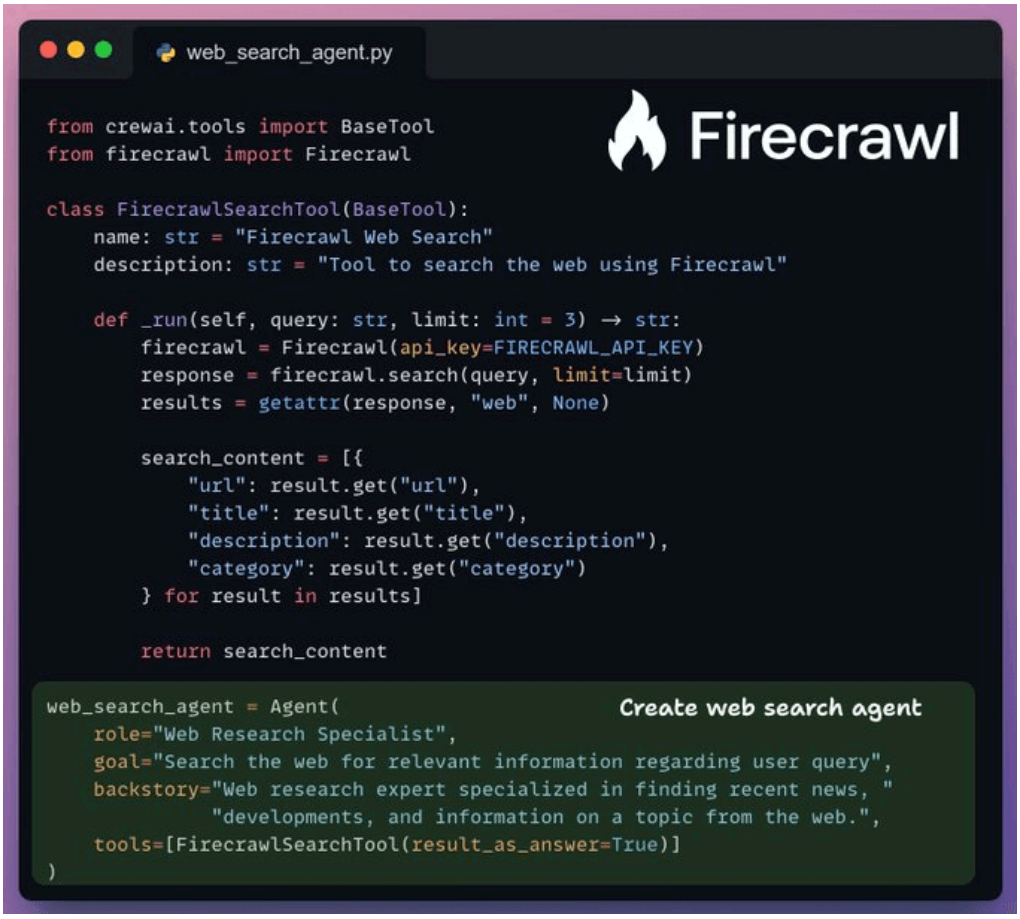
We use it to store and retrieve context from chat history and user data.



#5) Firecrawl web search

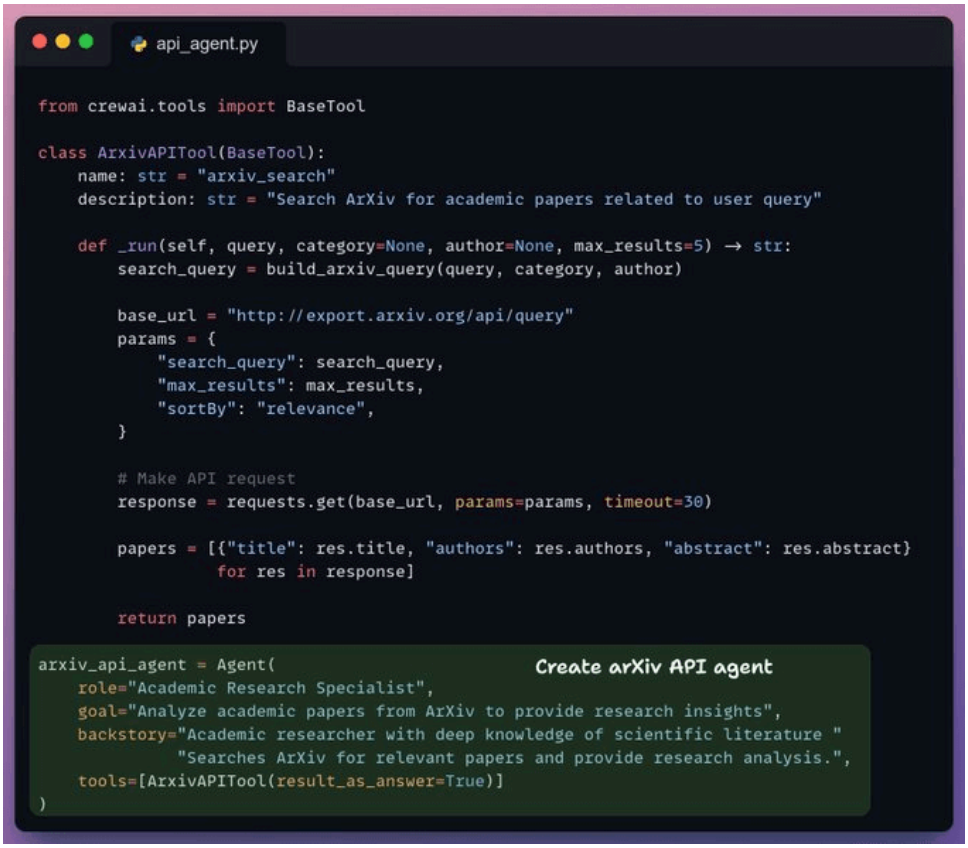
We use Firecrawl web search to fetch the latest news and developments related to the user query.

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



#6) ArXiv API search

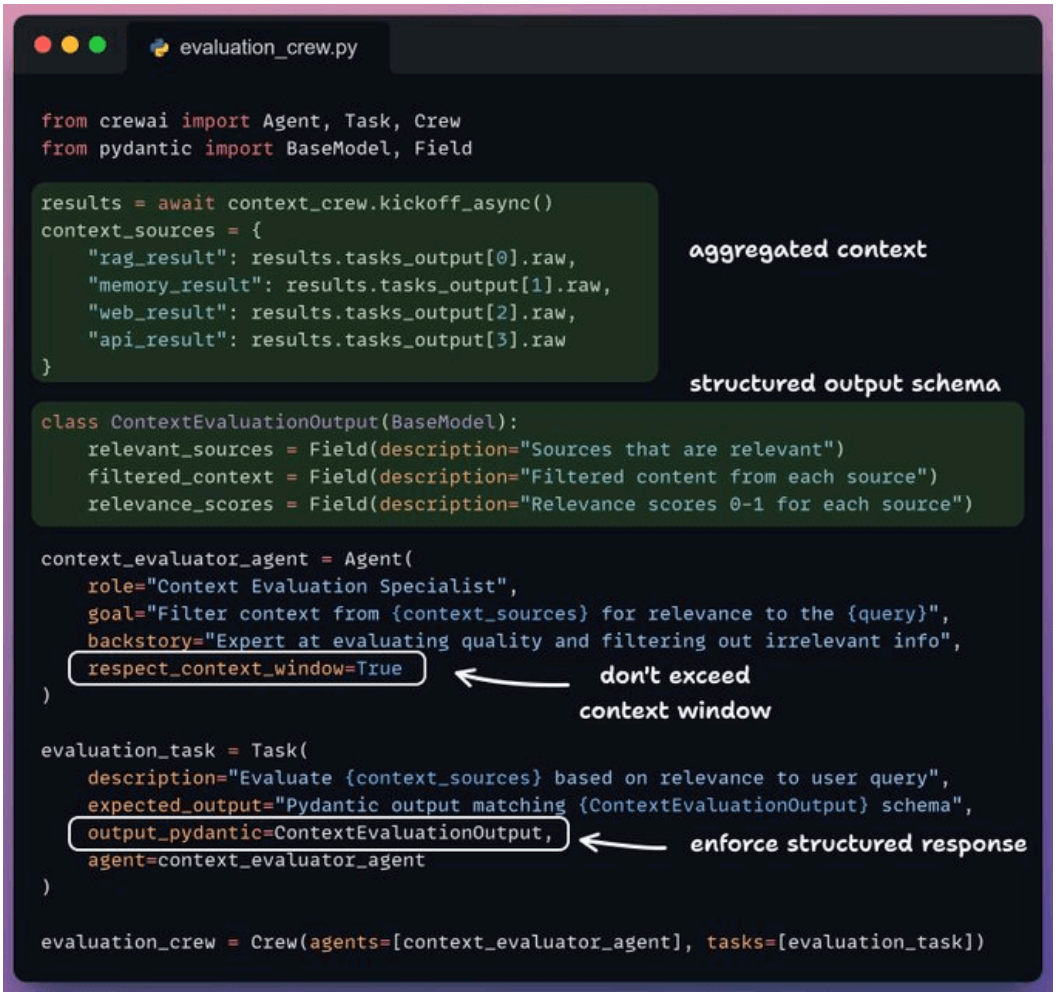
To further support research queries, we use the arXiv API to retrieve relevant results from their data repository based on the user query.



#7) Filter context

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

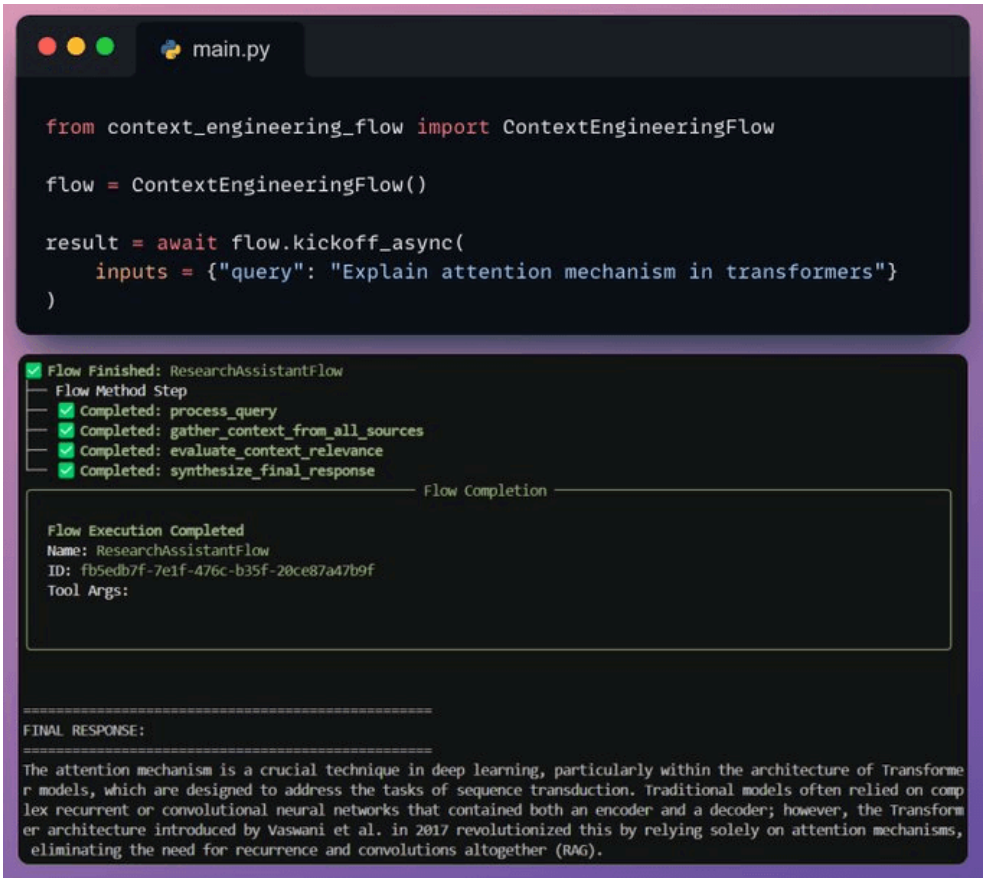
This filtered context is then passed to the synthesizer agent that generates the final response.



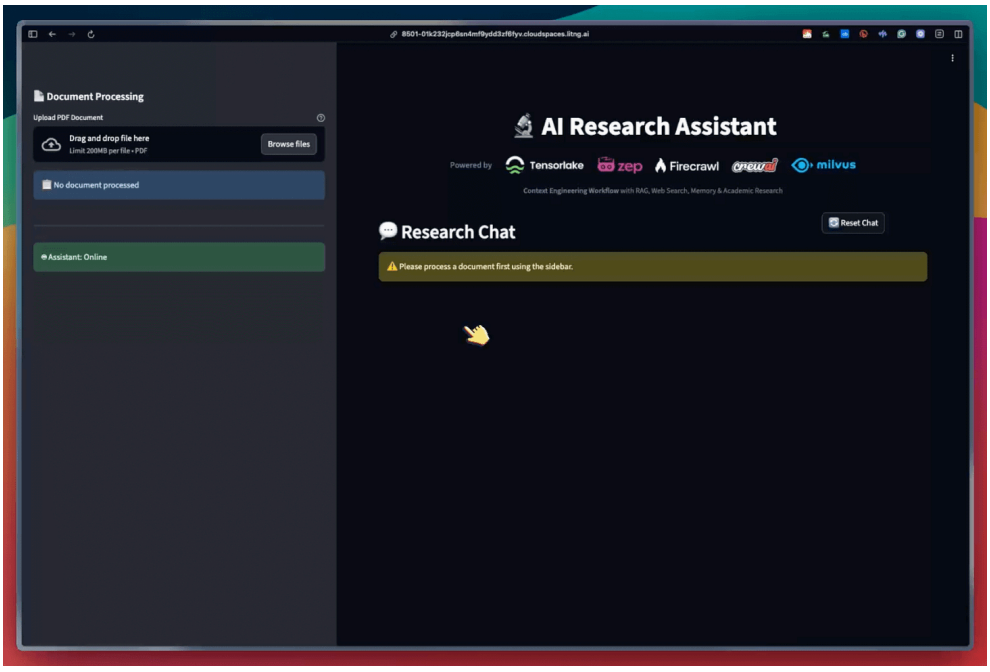
#8) Kick off the workflow

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

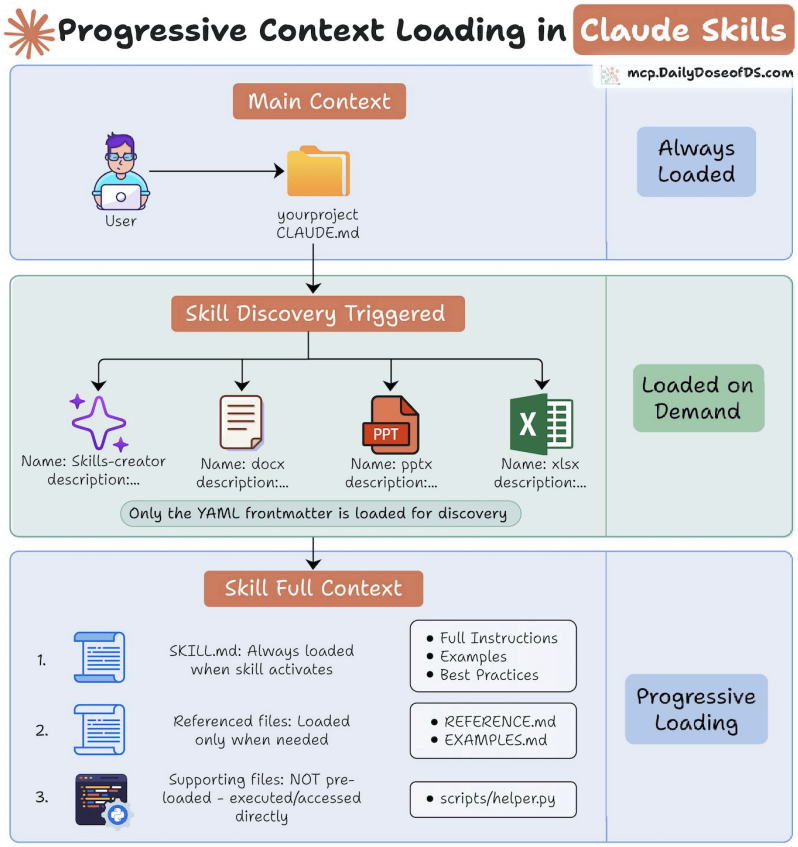
Claude Skills are Anthropic's mechanism for giving agents reusable, persistent abilities without overloading the model's context window.

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.

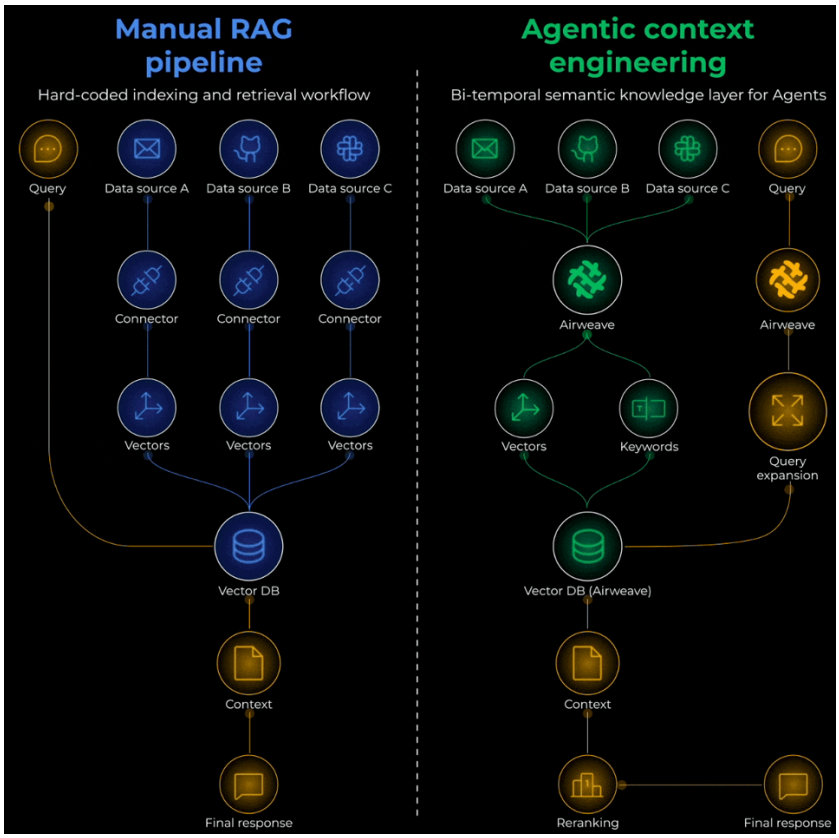
But the problem is that no real-world workflow looks like this.

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).

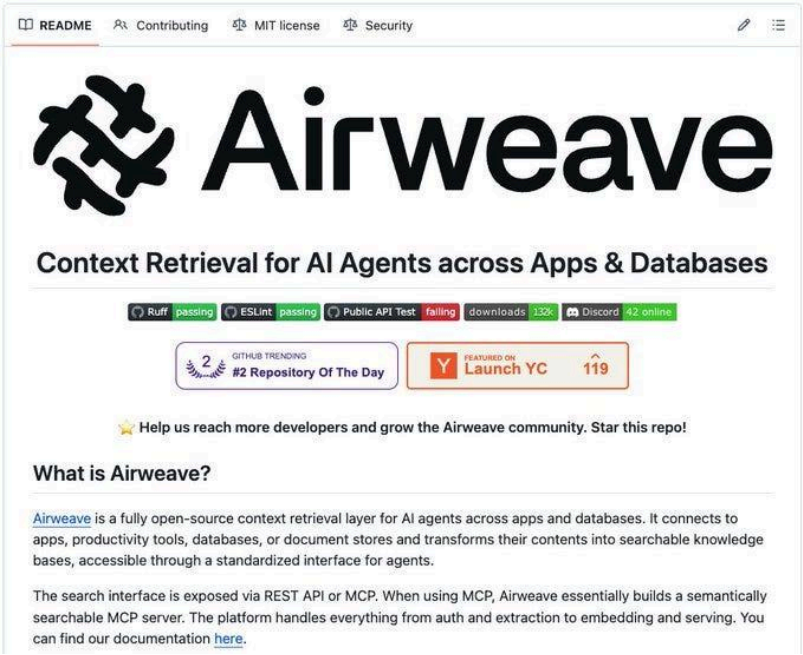
- Retrieval layer:
    - Expand vague queries to infer what users actually want.
    - Direct queries to the correct data sources.
    - Layer multiple search strategies like semantic-based, keyword-based, and graph-based.
    - Ensure retrieving only what users are authorized to see.
    - Weigh old 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:

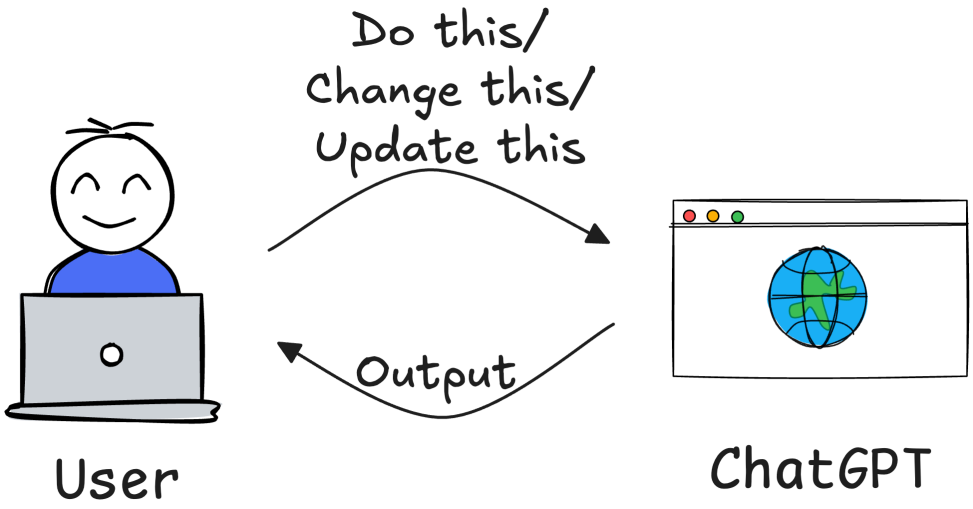
Context retrieval for Agents is an infrastructure problem, not an embedding problem.

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:

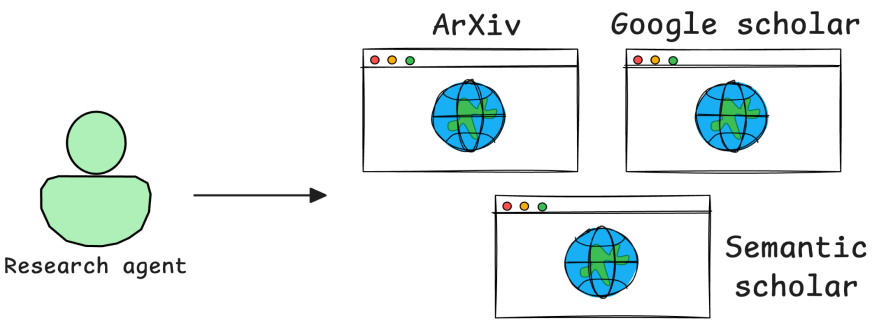


1. Ask for a summary of recent AI research papers.
2. Review the response and realize you need sources.
3. Obtain a list of papers along with citations.
4. Find that some sources are outdated, so you refine your query.
5. Finally, after multiple iterations, you get a useful output.

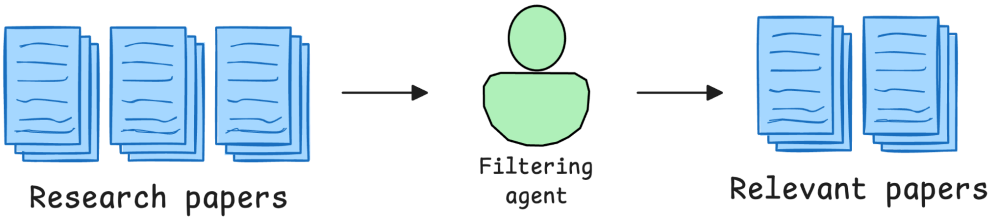
This iterative process takes time and effort, requiring you to act as the decision-maker at every step.

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

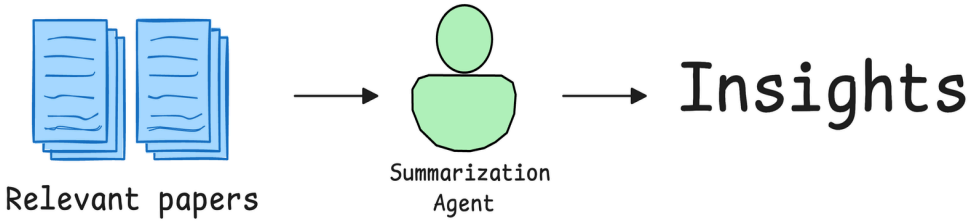
A Research Agent autonomously searches and retrieves relevant AI research papers from arXiv, Semantic Scholar, or Google Scholar.



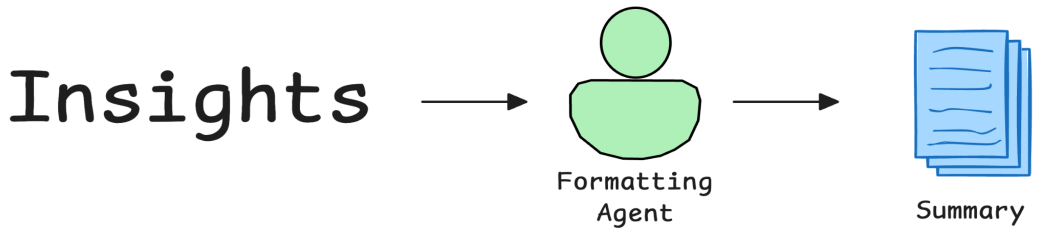
- A Filtering Agent scans the retrieved papers, identifying the most relevant ones based on citation count, publication date, and keywords.



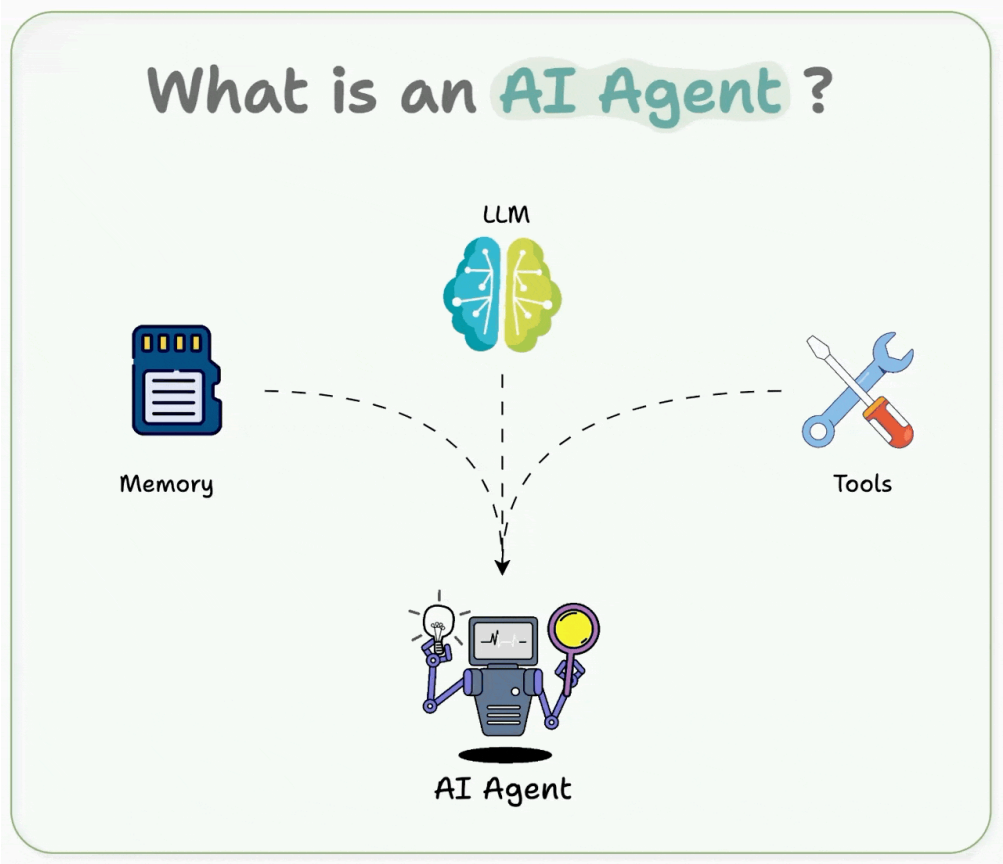
- A Summarization Agent extracts key insights and condenses them into an easy-to-read report.



- 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

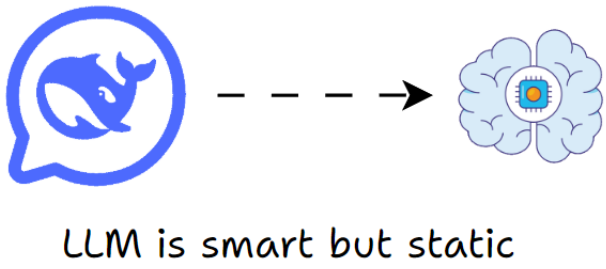
Let's break it down with a simple analogy:

- 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.

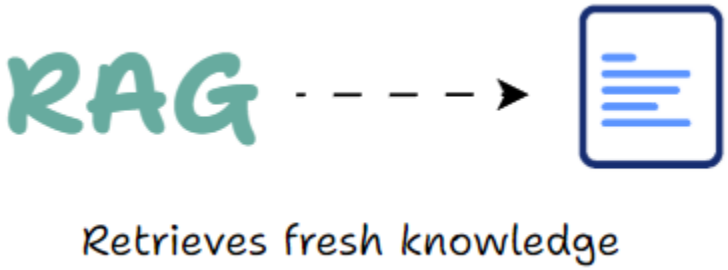
It can reason, generate, summarize but only using what it already knows (i.e., its training data).



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

RAG (Retrieval-Augmented Generation)

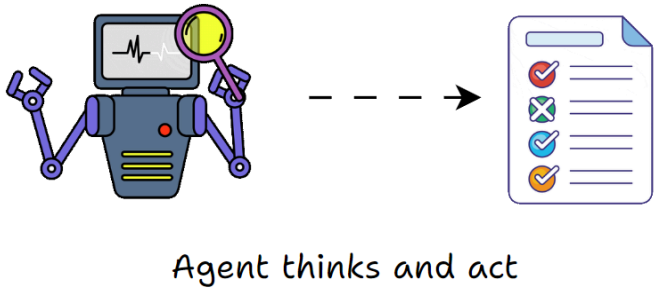
RAG enhances an LLM by retrieving external documents (from a vector DB, search engine, etc.) and feeding them into the LLM as context before generating a response.



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,

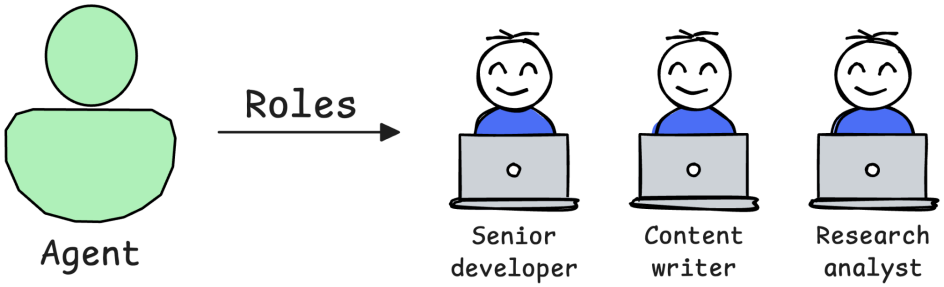
to be effective, they must be built with certain key principles in mind. There are six essential building blocks that make AI agents more reliable, intelligent, and useful in real-world applications:

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.

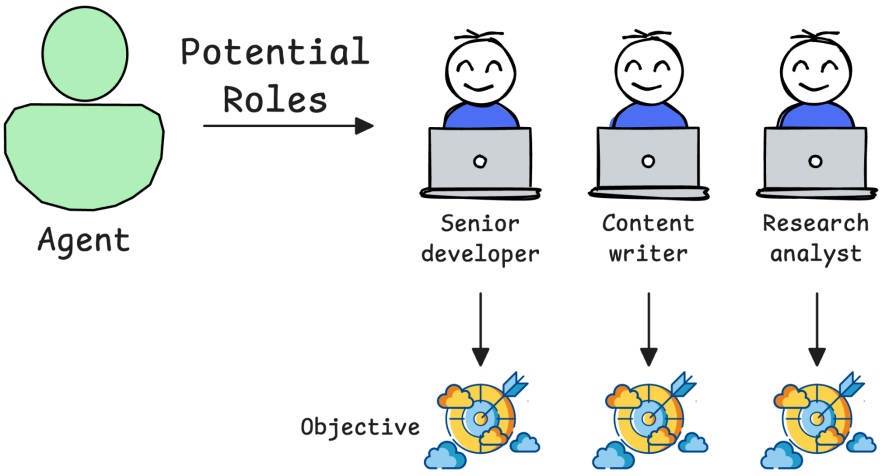
Why?

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.

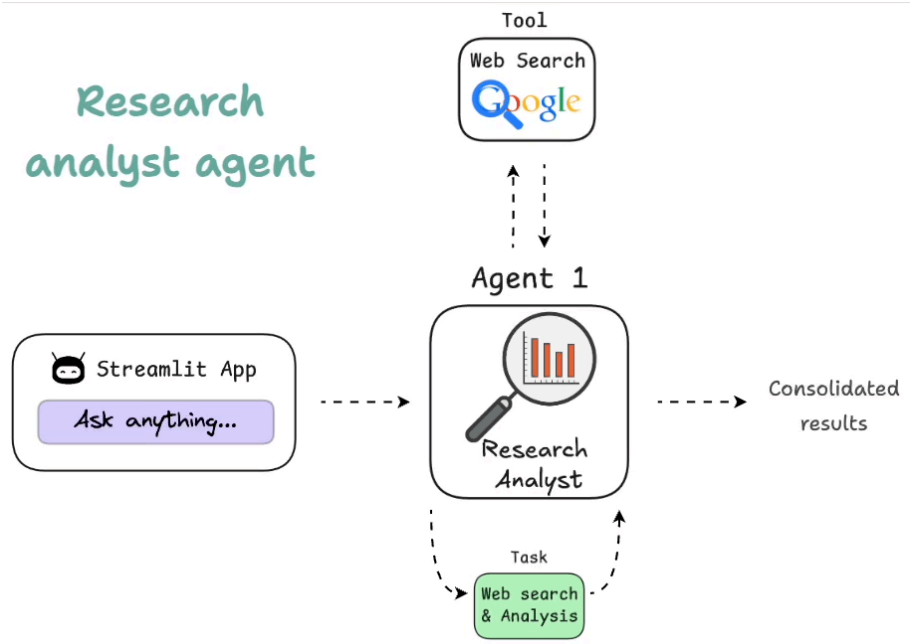
For example, a marketing agent should stick to messaging, tone, and audience not pricing or market analysis.

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.

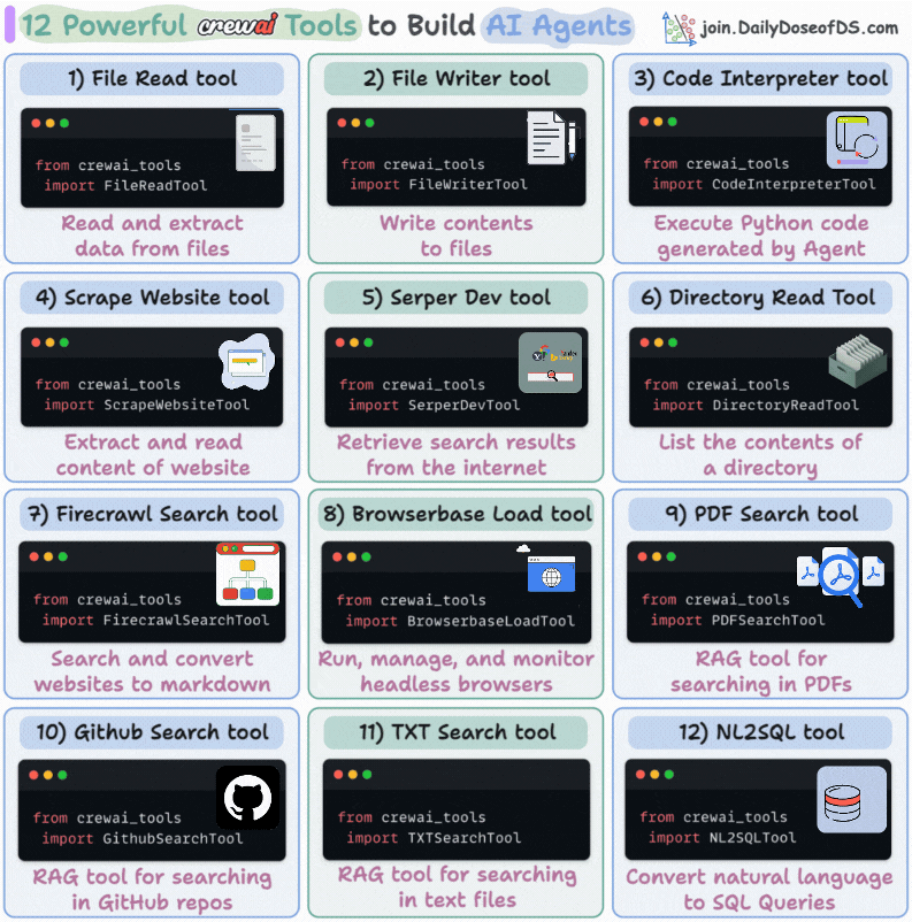
#3.1) Custom tools

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:

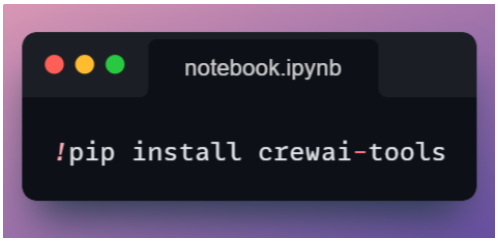


However, you may need to build custom tools at times.

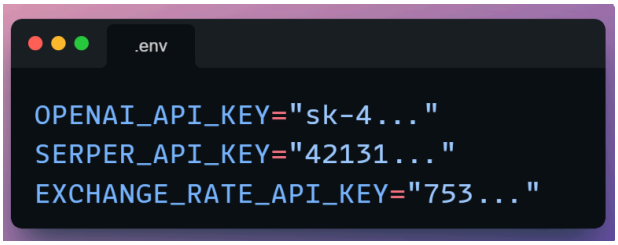
In this example, we're building a real-time currency conversion tool inside CrewAI. Instead of making an LLM guess exchange rates, we integrate a custom tool that fetches live exchange rates from an external API and provides some insights.

Below, let's look at how you can build one for your custom needs in the CrewAI framework.

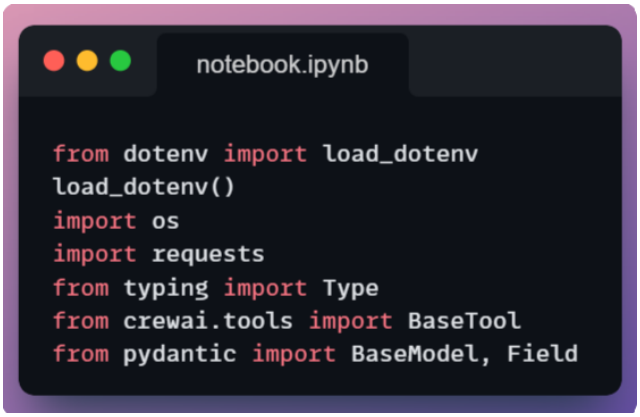
Firstly, make sure the tools package is installed:



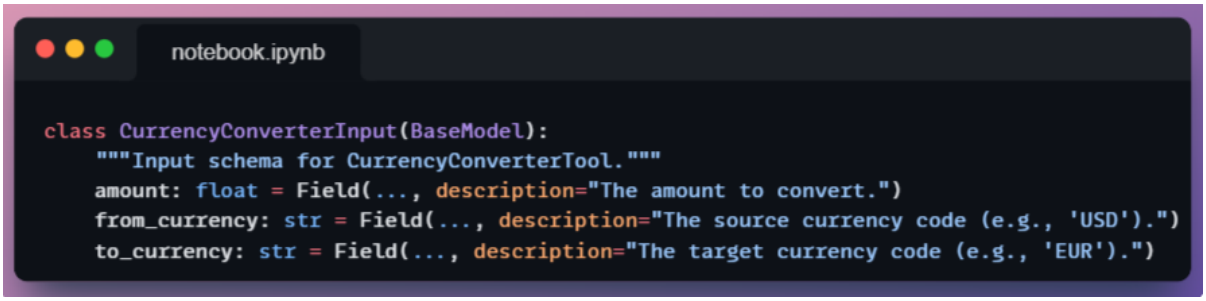
You would also need an API key from here: https://www.exchangerate-api.com/ (it's free). Specify it in the .env file as shown below:



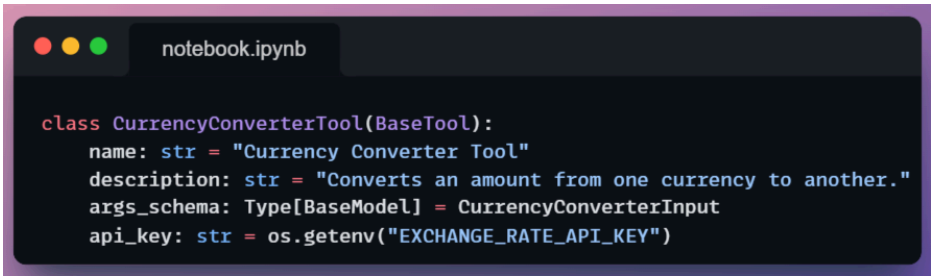
Once that's done, we start with some standard import statements:



Next, we define the input fields the tool expects using Pydantic.

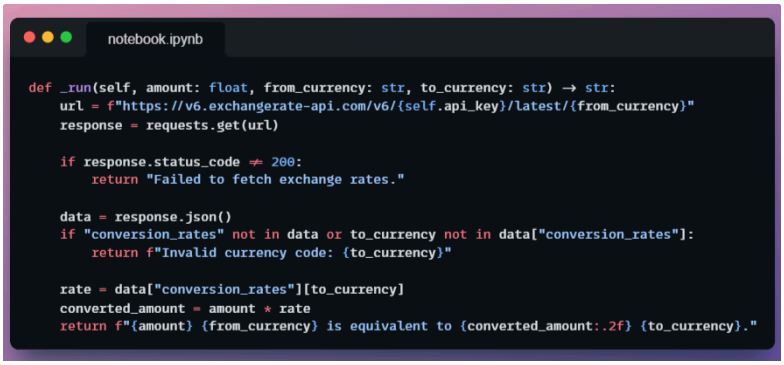


Now, we define the CurrencyConverterTool by inheriting from *BaseTool:*



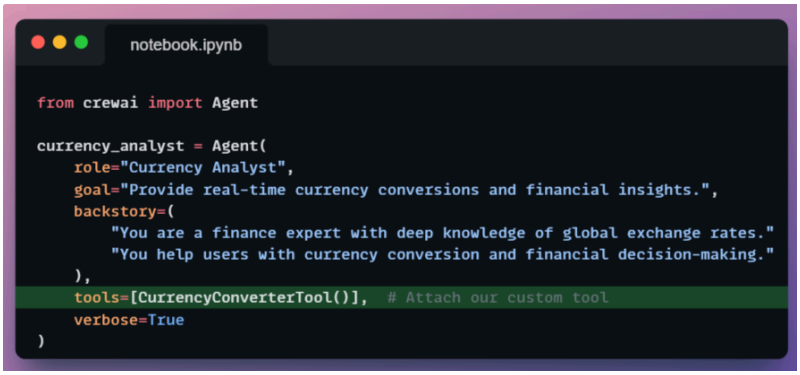
Every tool class should have the _run method which we will execute whenever the Agents wants to make use of it.

For our use case, we implement it as follows:

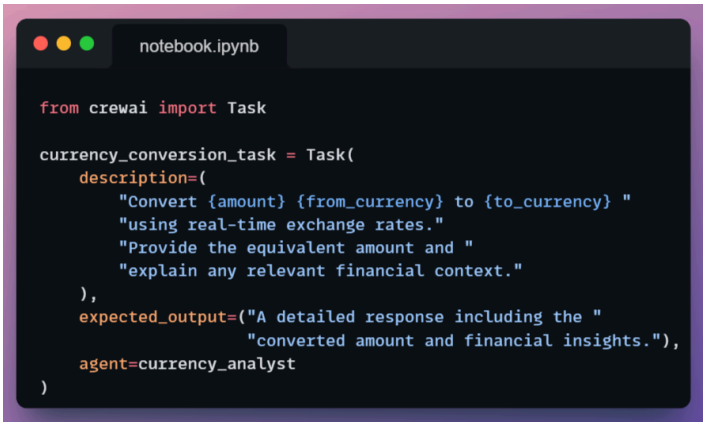


In the above code, we fetch live exchange rates using an API request. We also handle errors if the request fails or the currency code is invalid.

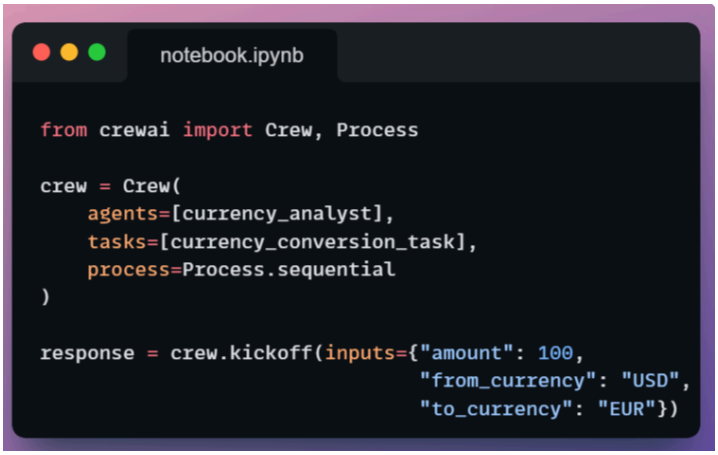
Now, we define an agent that uses the tool for real-time currency analysis and attach our CurrencyConverterTool, allowing the agent to call it directly if needed:



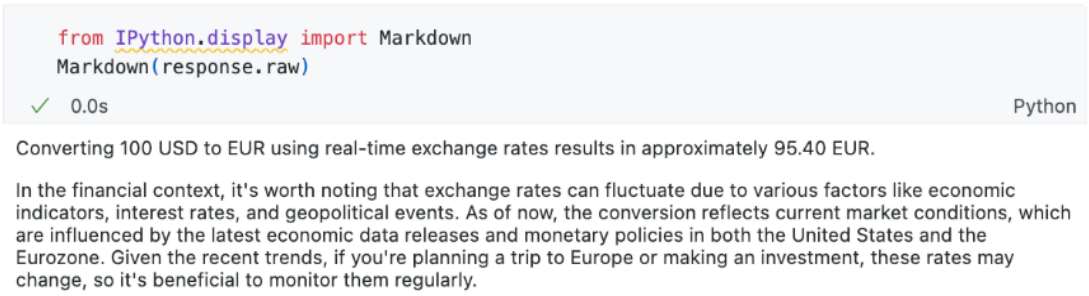
We assign a task to the currency_analyst agent.



Finally, we create a Crew, assign the agent to the task, and execute it.



Printing the response, we get the following output:



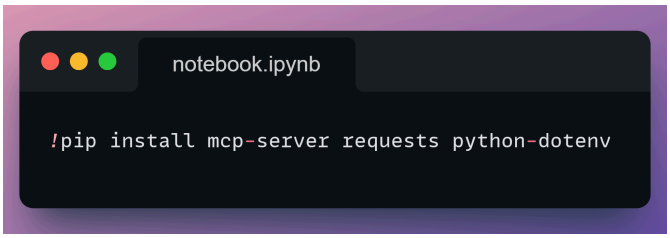
Works as expected!

#3.2) Custom tools via MCP

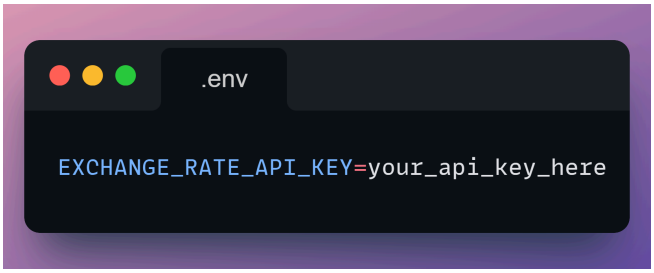
Now, let's take it a step further.

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:

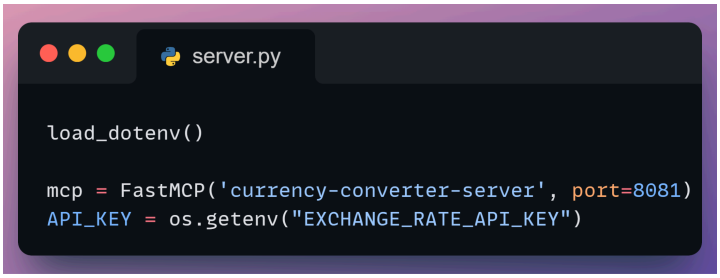


We'll continue using ExchangeRate-API in our .env file:

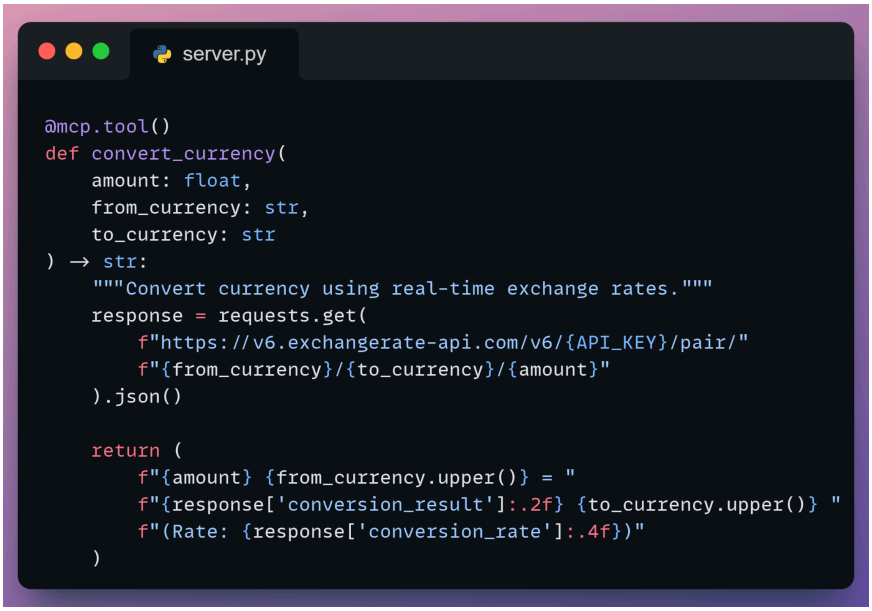


We'll now write a lightweight server.py script that exposes the currency converter tool. We start with the standard imports:

Now, we load environment variables and initialize the server:

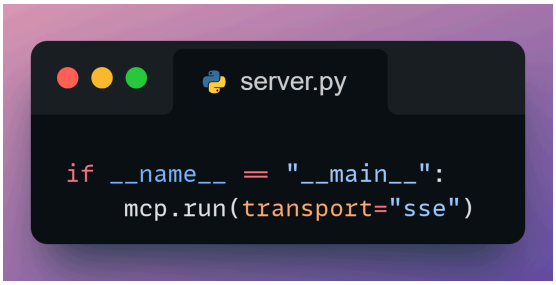


Next, we define the tool logic with @mcp.tool():



This function takes three inputs - amount, source currency, and target currency and returns the converted result using the real-time exchange rate API.

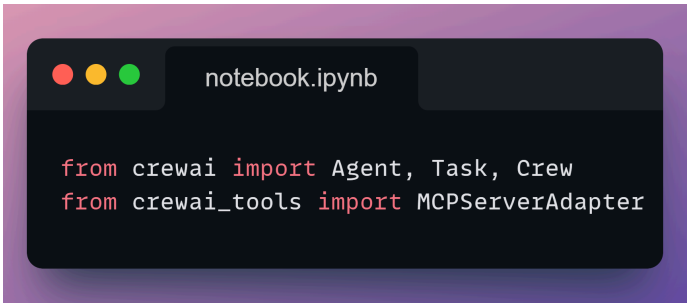
To make the tool accessible, we need to run the MCP server. Add this at the end of your script:



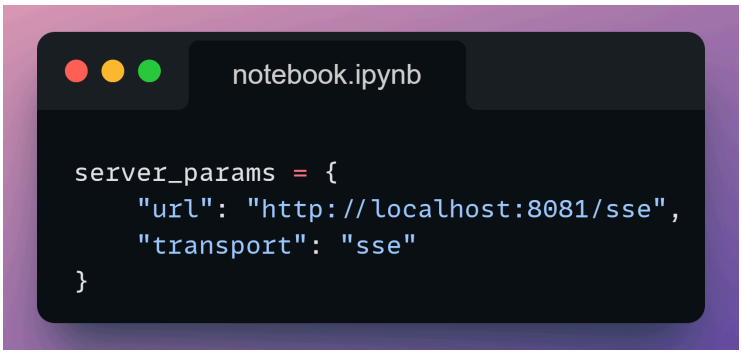
This starts the server and exposes your convert_currency tool at: http://localhost:8081/sse.

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.

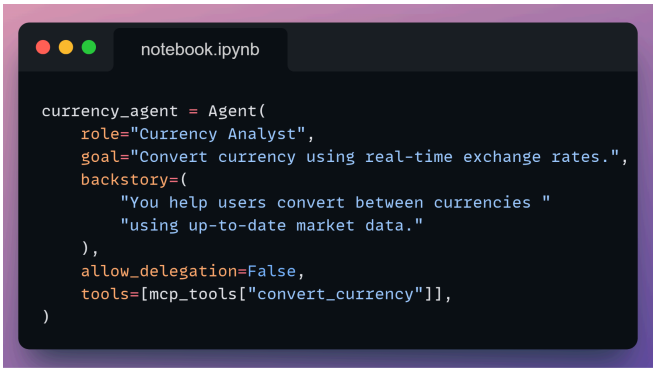


Next, we connect to the MCP tool server. Define the server parameters to connect to your running tool (from server.py).

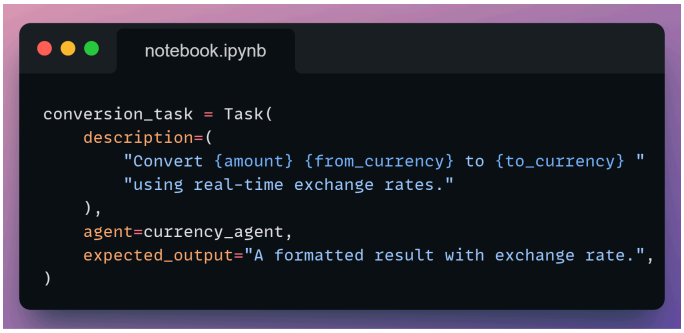


Now, we use the discovered MCP tool in an agent:

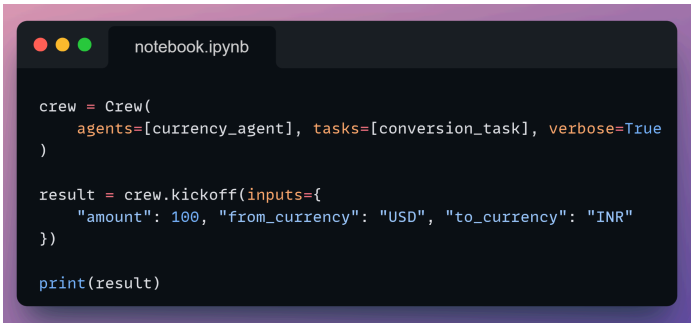
This agent is assigned the convert_currency tool from the remote server. It can now call the tool just like a locally defined one.



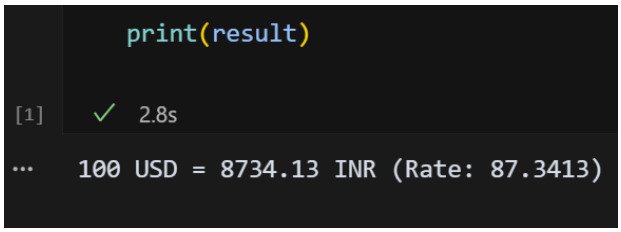
We give the agent a task description:



Finally, we create the Crew, pass in the inputs and run it:



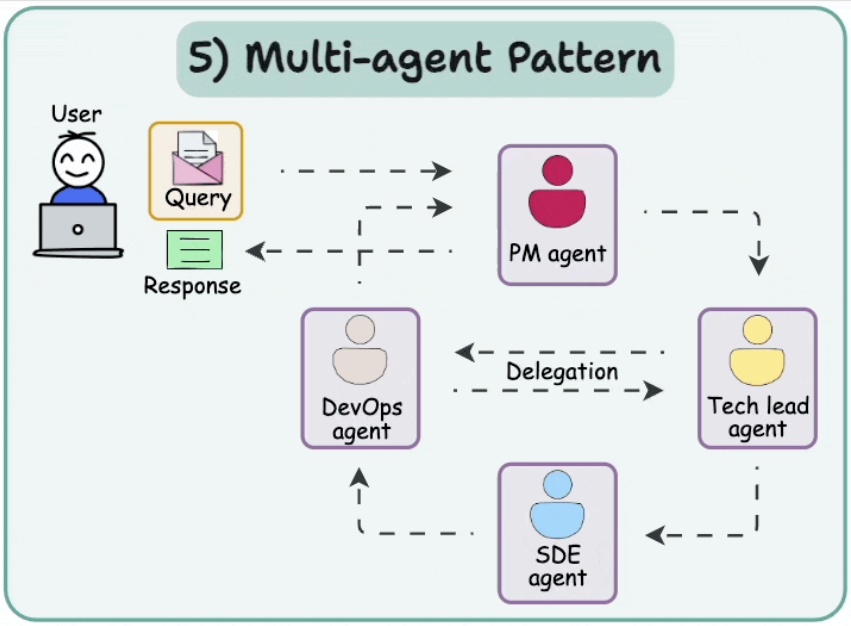
Printing the result, we get the following output:



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.

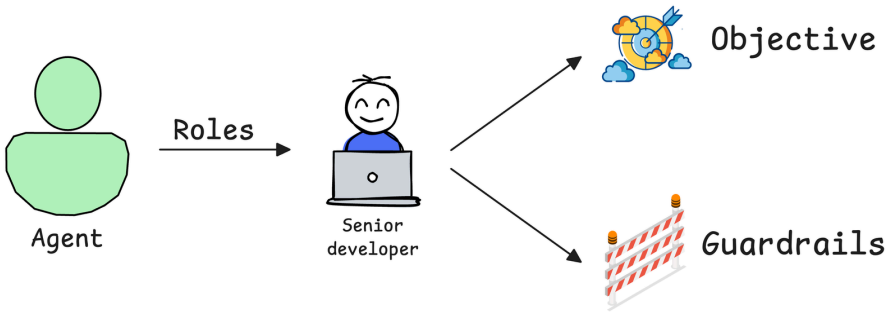
The best practice is to enable agent collaboration by designing workflows where agents can exchange insights and refine their responses together.

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:

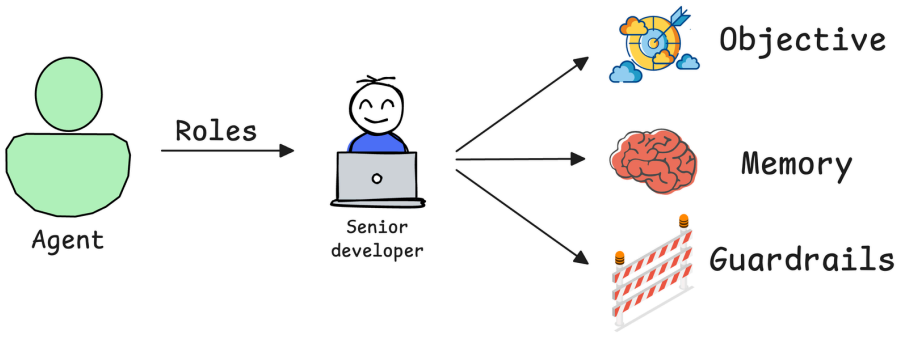
- Limiting tool usage: Prevent an agent from overusing APIs or generating irrelevant queries.
- Setting validation checkpoints: Ensure outputs meet predefined criteria before moving to the next step.
- Establishing fallback mechanisms: If an agent fails to complete a task, another agent or human reviewer 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).

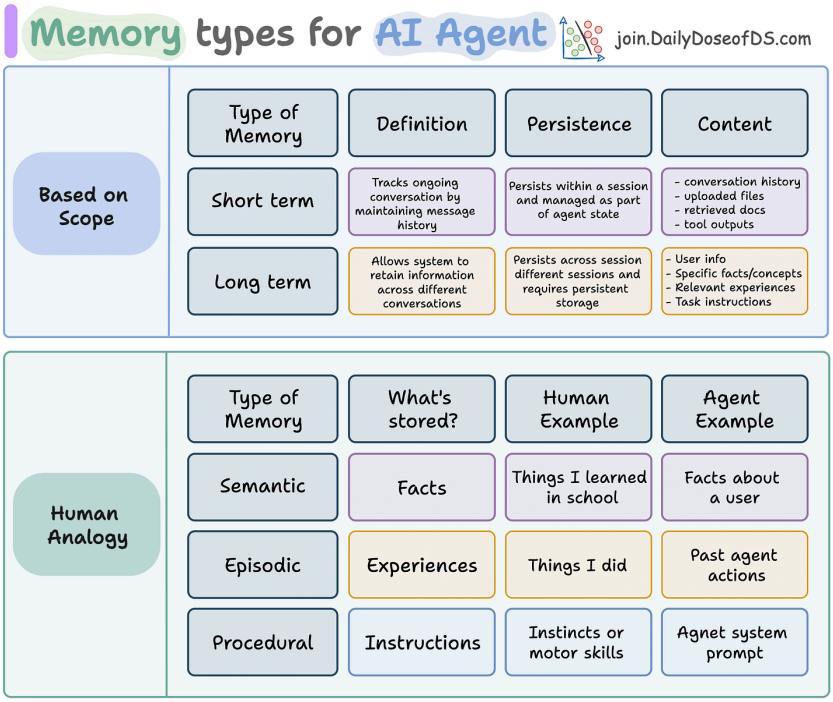
For example, in an AI-powered tutoring system, memory allows the agent to recall past lessons, tailor feedback, and avoid repetition.

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)



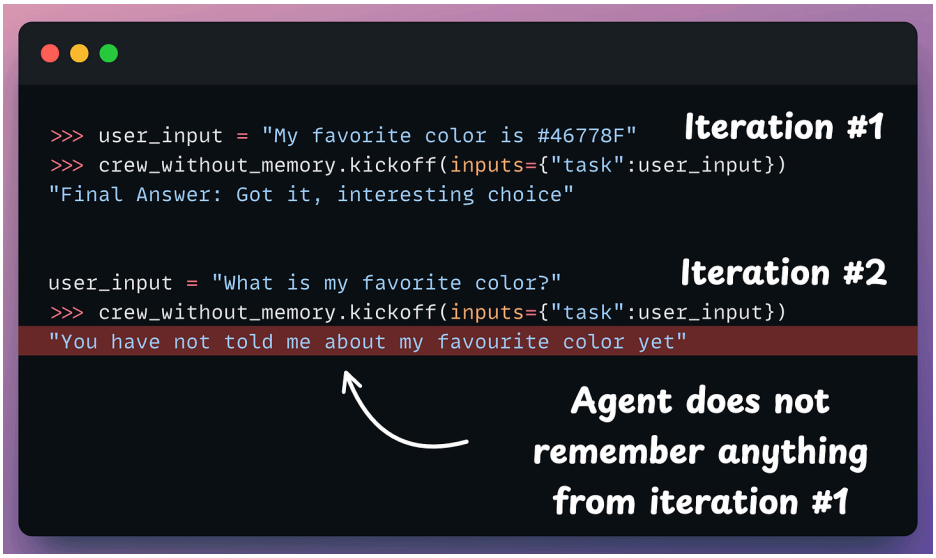
This memory isn't just nice-to-have but it enables agents to learn from past interactions without retraining the model.

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):



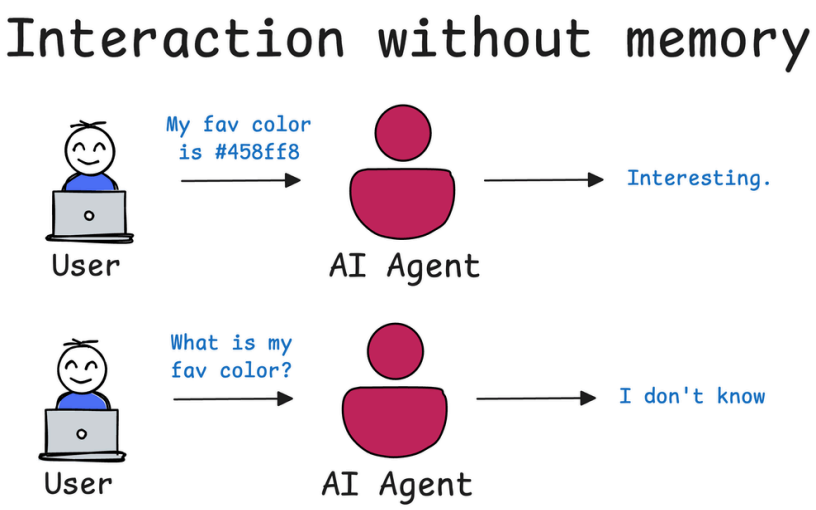
- In iteration #1, the user mentions their favorite color.
- In iteration #2, the Agent knows nothing about iteration #1.

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.

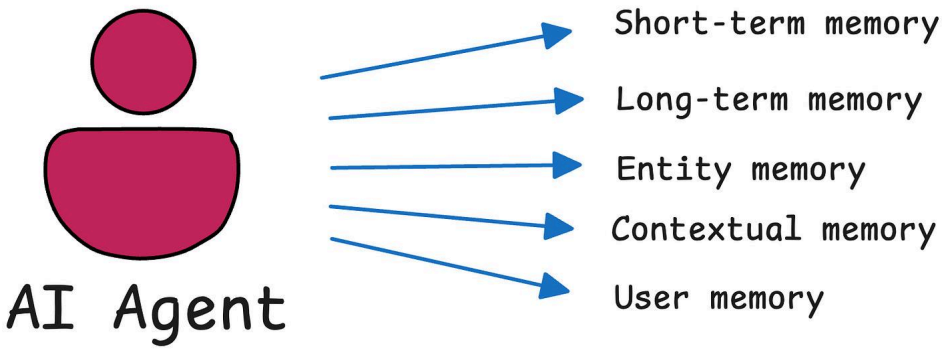


It doesn't matter if the user told the Agent their name five seconds ago, it's forgotten. If the Agent helped troubleshoot an issue in the last session, it won't remember any of it now.

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.

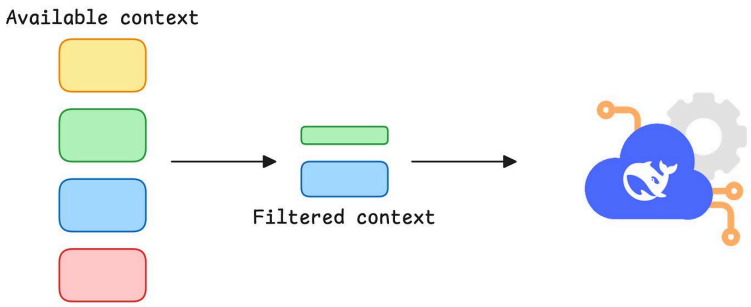


- Short-Term Memory
- Long-Term Memory
- Entity Memory
- Contextual Memory, and

- 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.