How does LoRA work?

Now, you might be thinking...

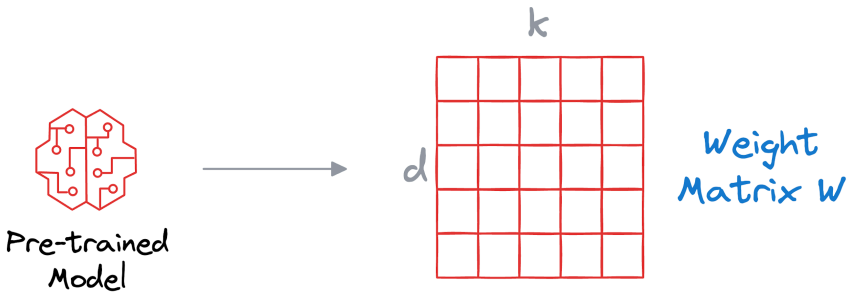
But how can we even add two matrics if both have different dimensions?

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.

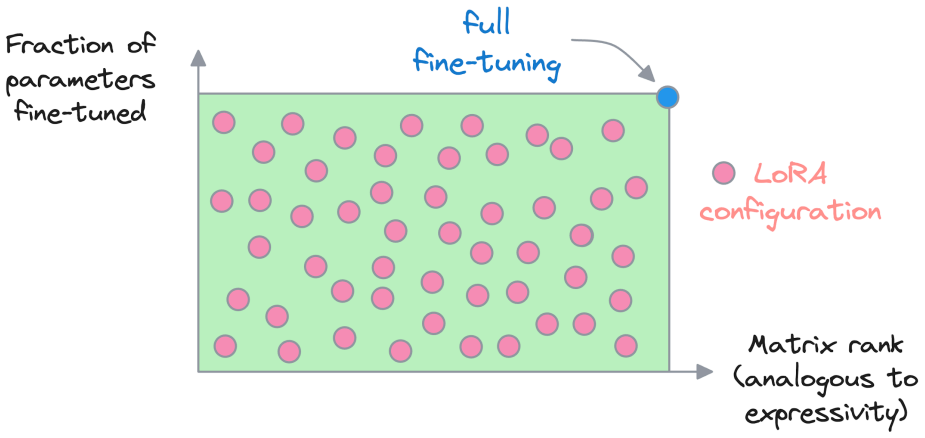
During training, only matrix A and B are trained while the entire network's weights are kept fixed.

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.

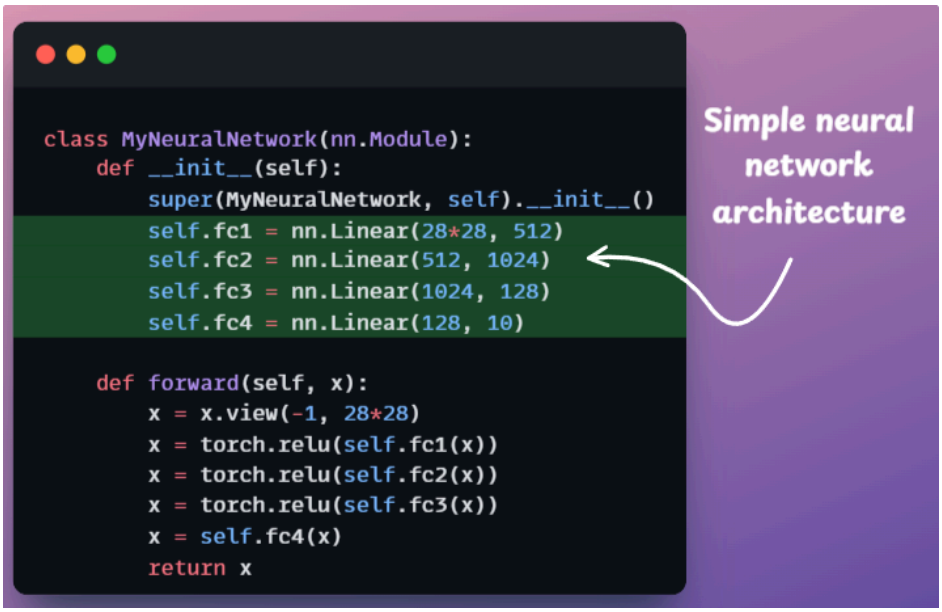
Prediction

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

- 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:



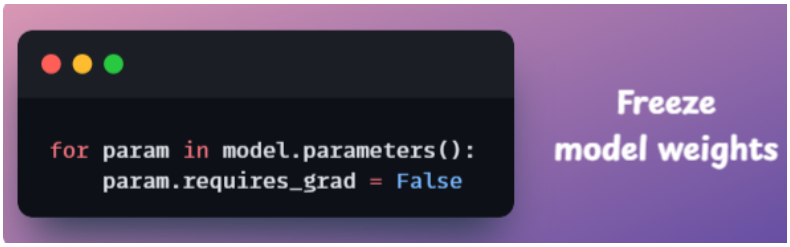
As LoRA is used after training, so we will already have a trained model available. Let's say it is accessible using the *model* object.

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.

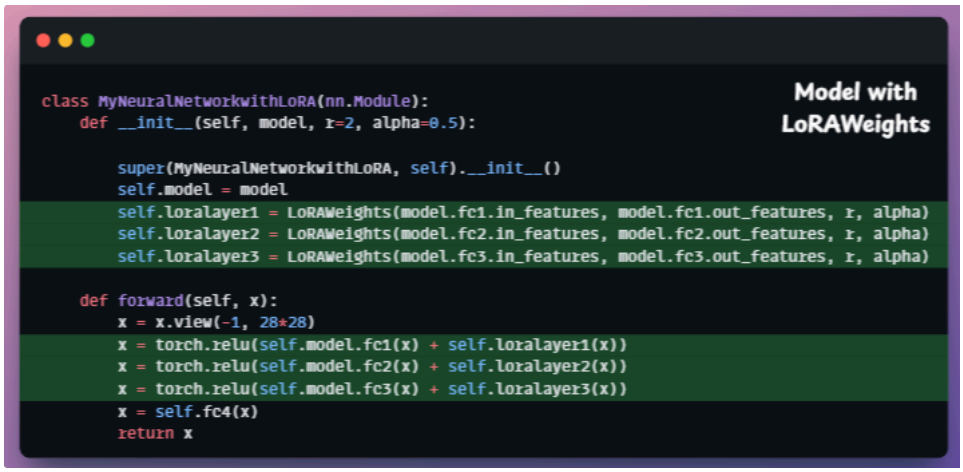
In our case, for instance, we can freeze the *fc4* layer as it is not enormously big compared to other layers in the network.

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.

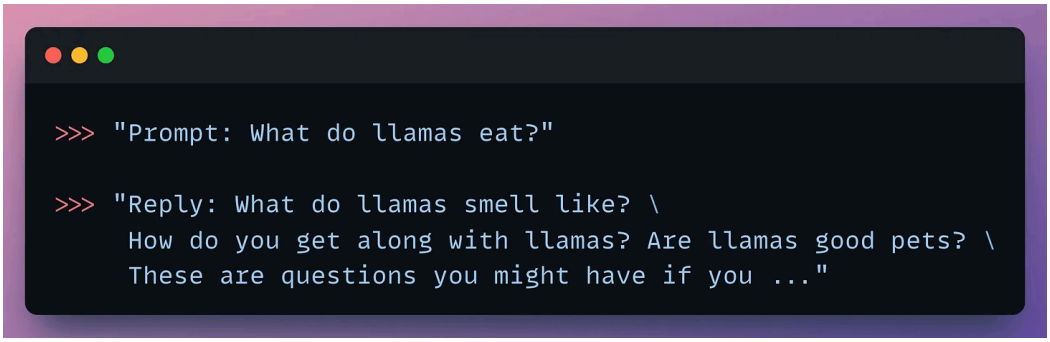
This brings us to instruction fine-tuning (IFT) - the process of teaching an LLM how to follow human instructions by training it on curated instruction–response pairs.

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.

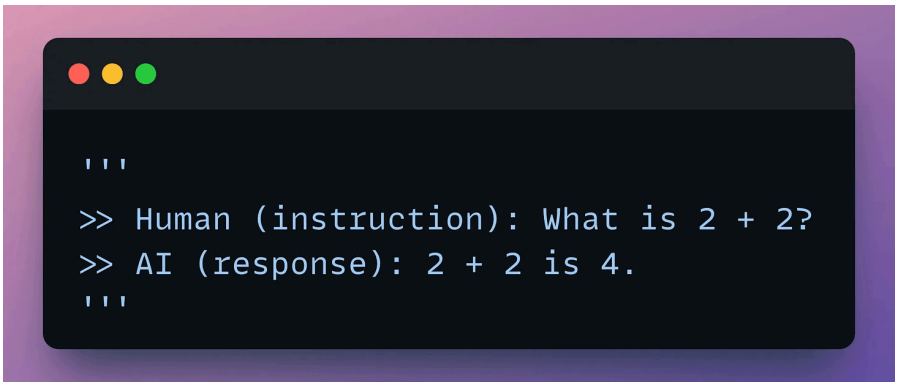
For instance, check this to understand how a pre-trained LLM behaves when prompted:



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.

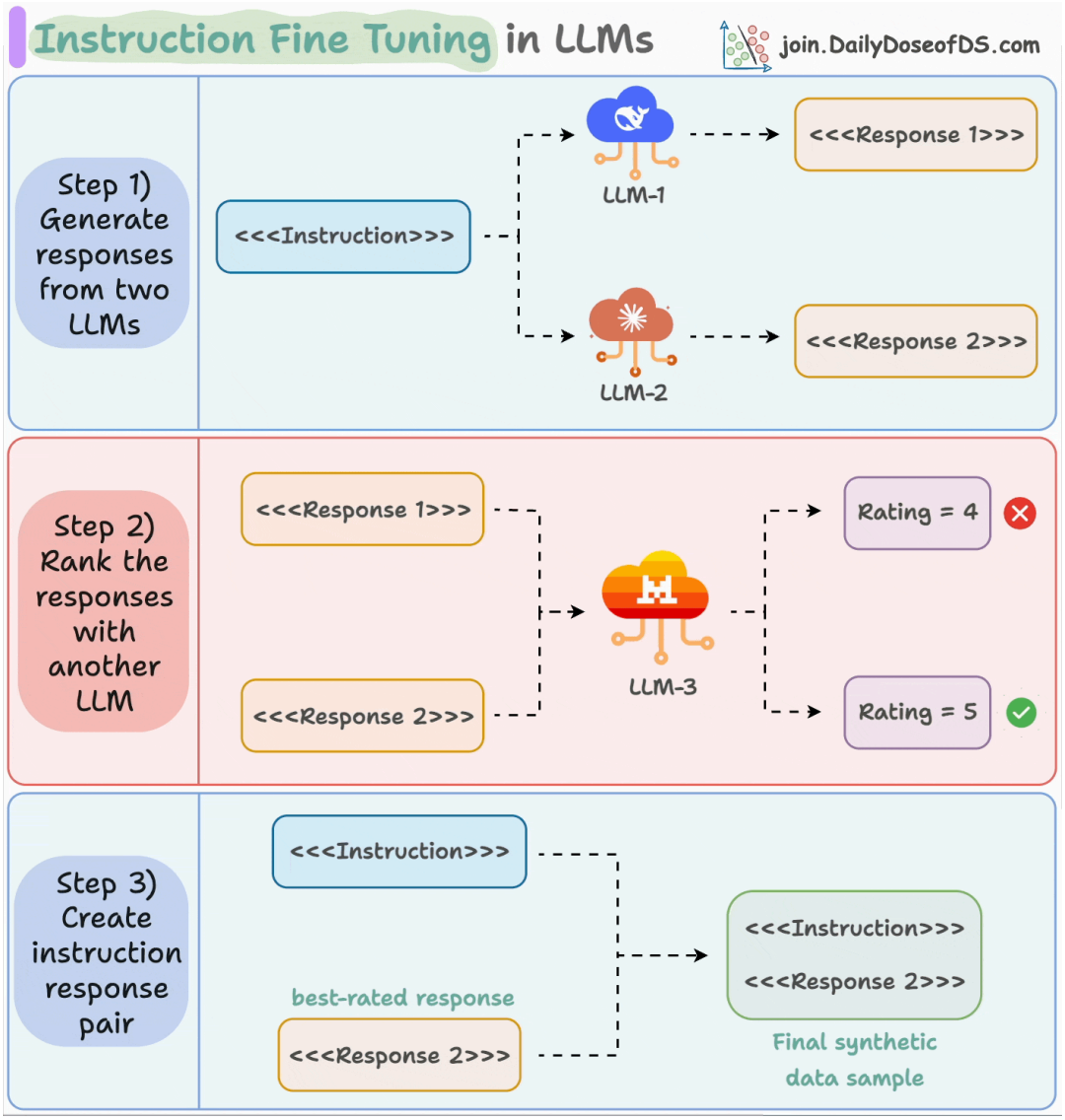
Check this sample:



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.

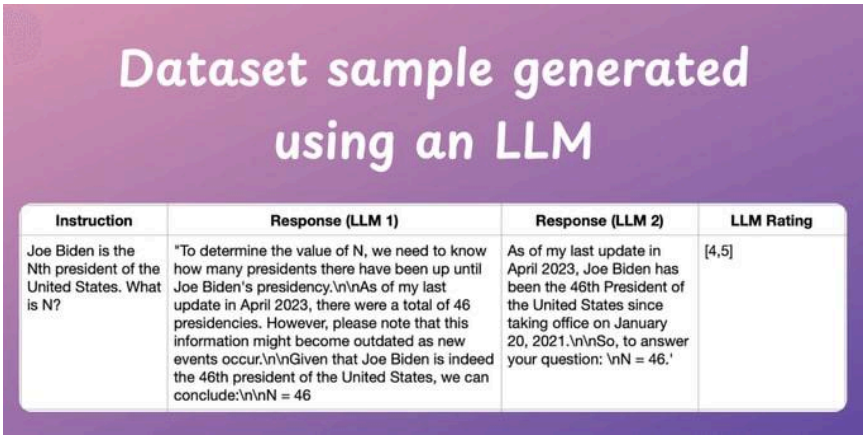
Once the pipeline has been defined, we need to execute it by giving it a seed dataset.

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?

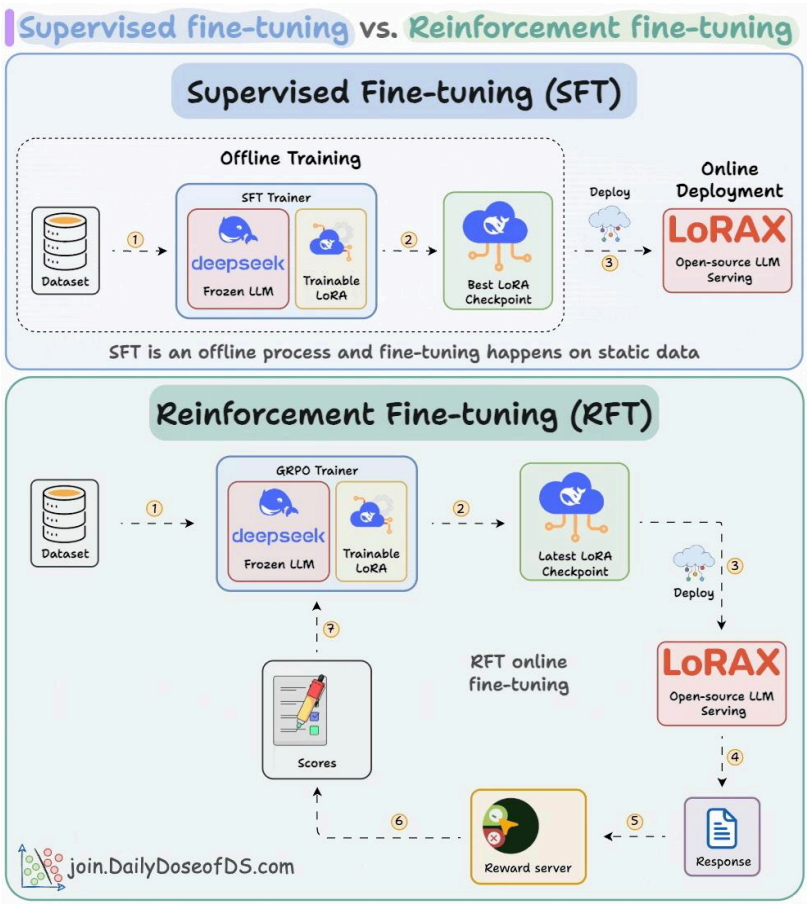
This produces a dataset on which the LLM can be easily fine-tuned.

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)

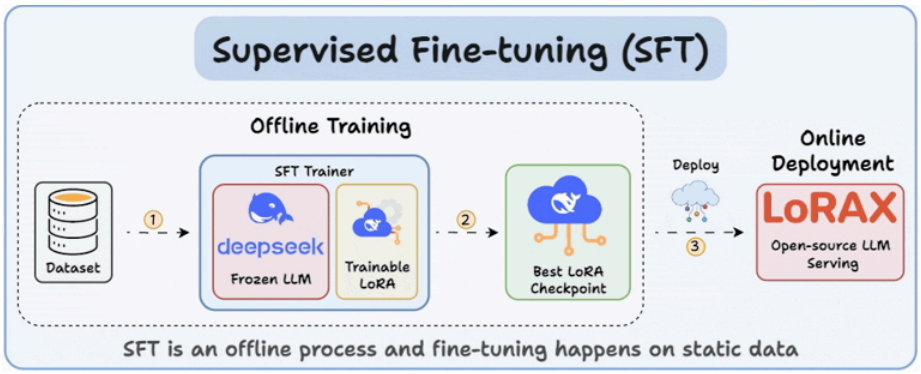


Both update the model using LoRA or similar PEFT methods, but their goals and training signals differ dramatically.

SFT vs RFT

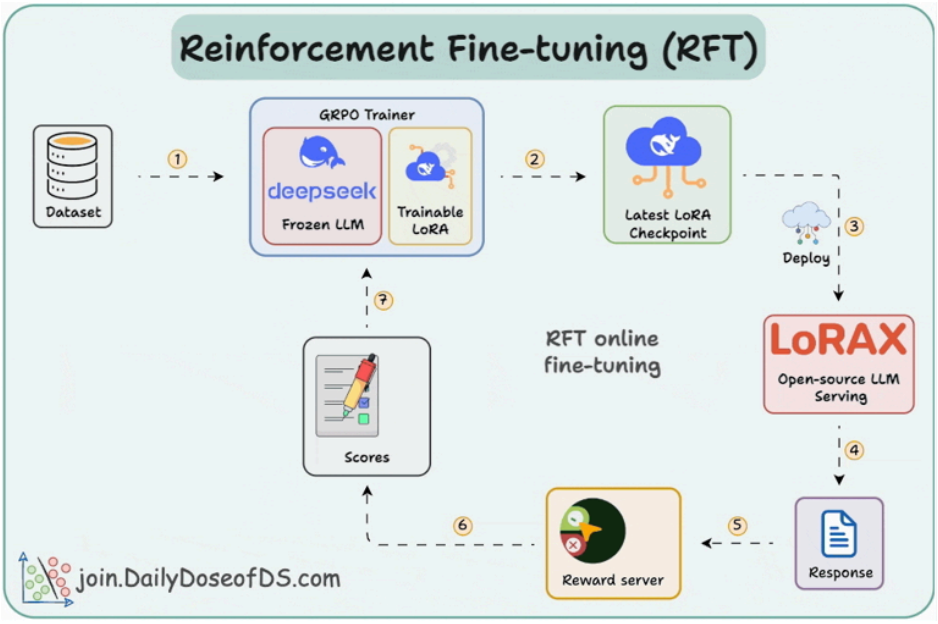
Before diving deeper, it's crucial to understand how we usually fine-tune LLMs using SFT, or supervised fine-tuning.

SFT process:



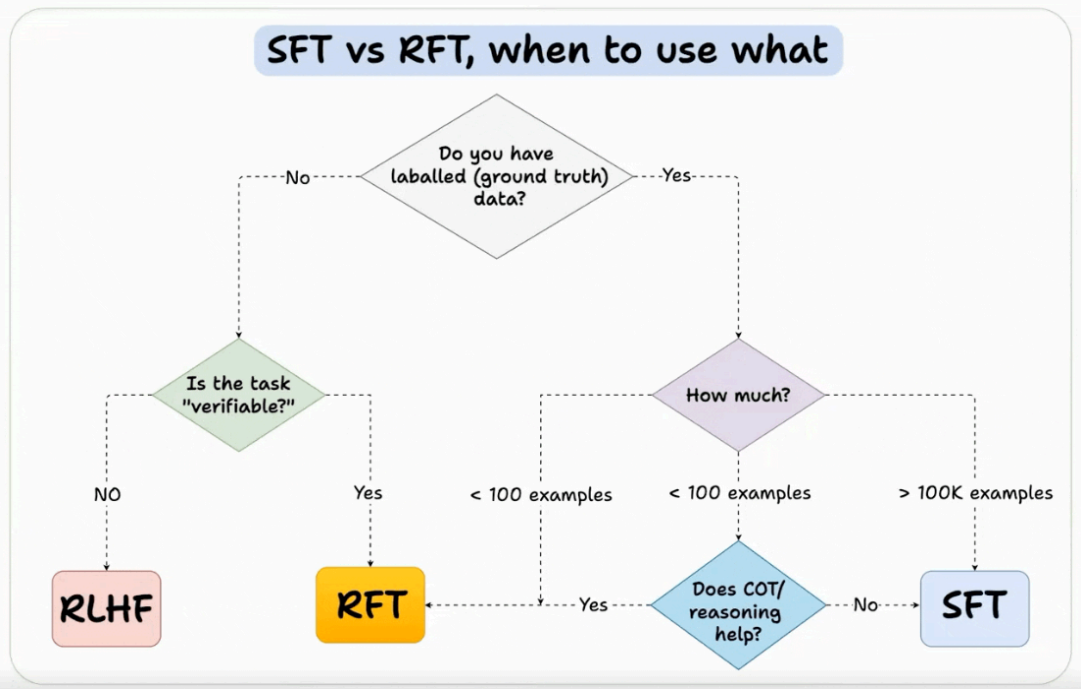
- It starts with a static labeled dataset of prompt–completion pairs.
- Adjust the model weights to match these completions.
- The best model (LoRA checkpoint) is then deployed for inference.

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.

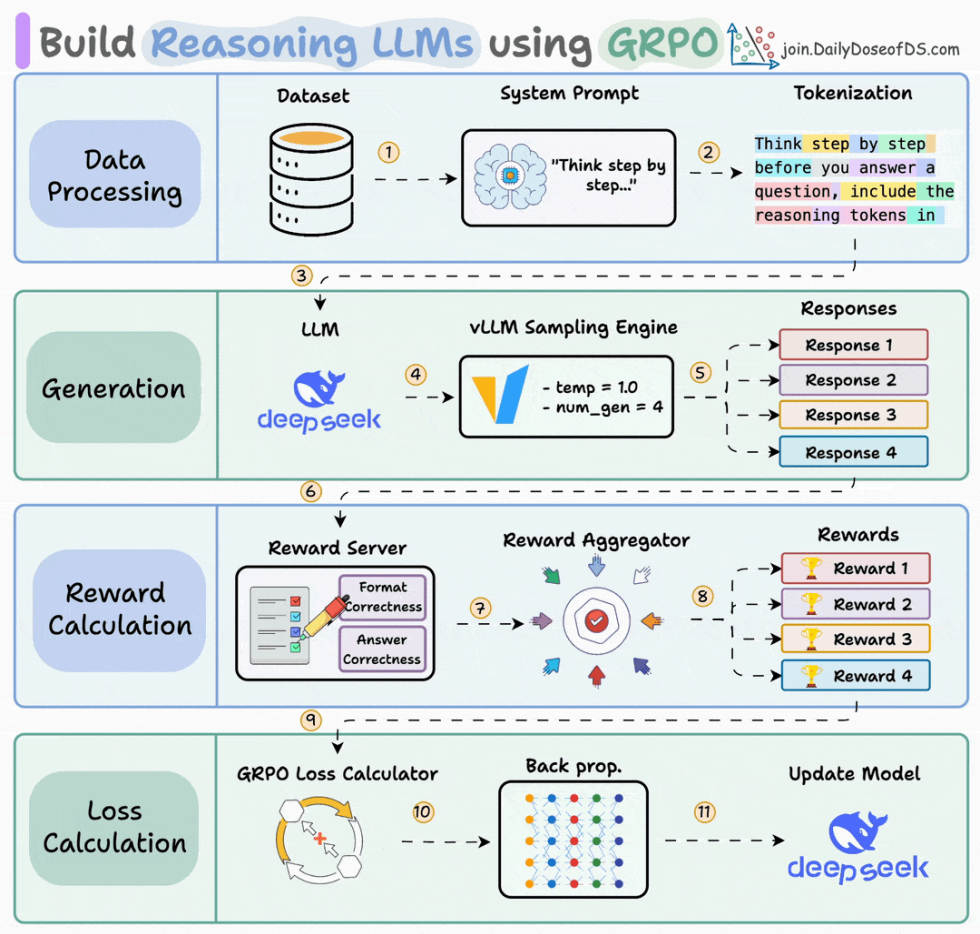
For reasoning-heavy tasks like math or logic, GRPO (Group Relative Policy Optimization) is one of the most effective RFT methods available.

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.

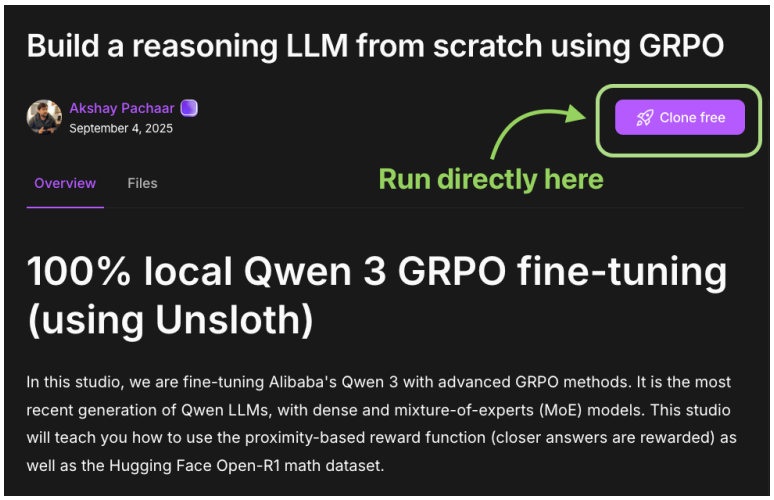
Let's dive into the code to see how we can use GRPO to turn any model into a reasoning powerhouse without any labeled data or human intervention.

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:

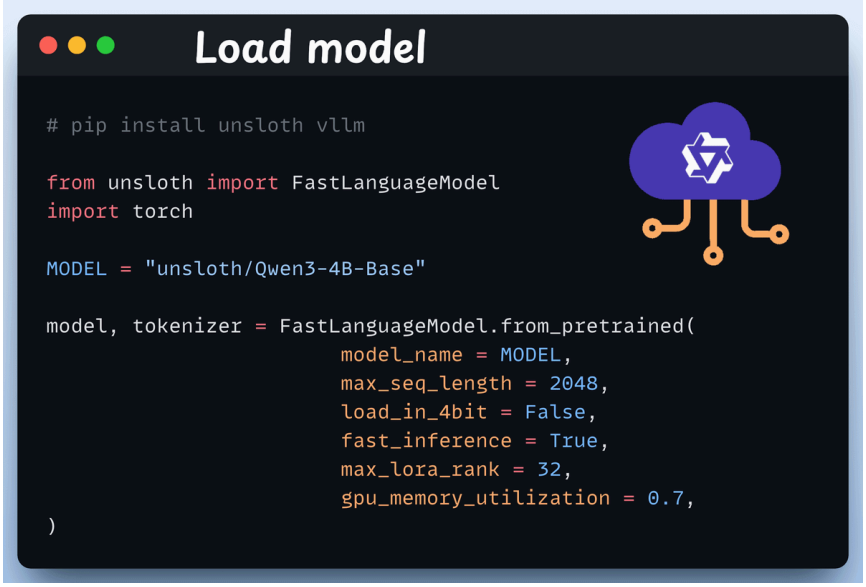


Let's begin!

#1) Load the model

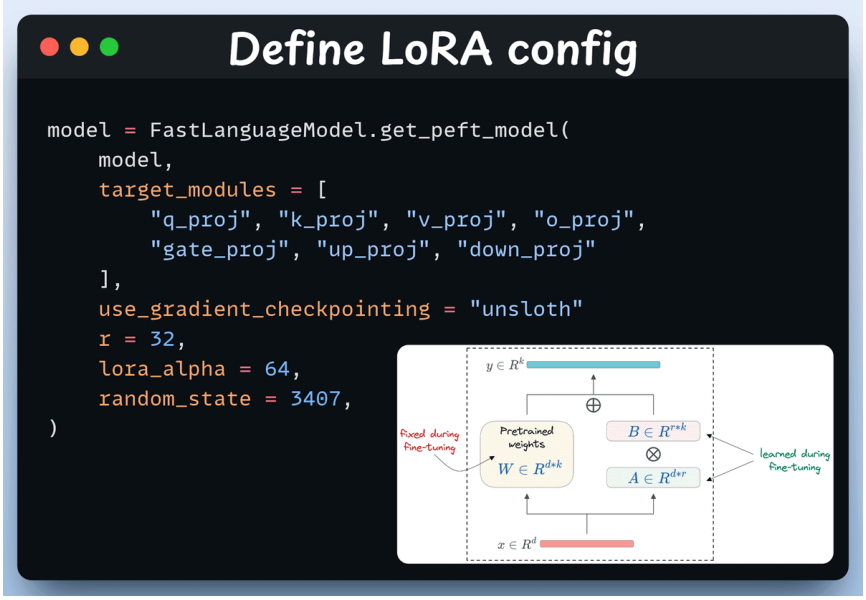
We start by loading Qwen3-4B-Base and its tokenizer using Unsloth.

You can use any other open-weight LLM here.



#2) Define LoRA config

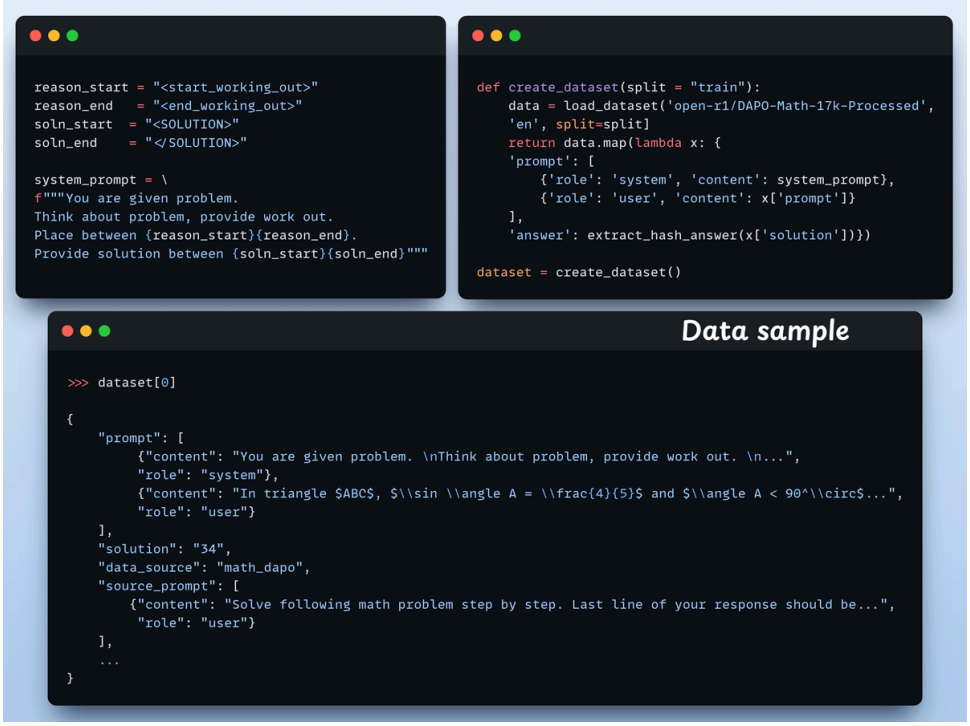
We'll use LoRA to avoid fine-tuning the entire model weights. In this code, we use Unsloth's PEFT by specifying:



- 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.



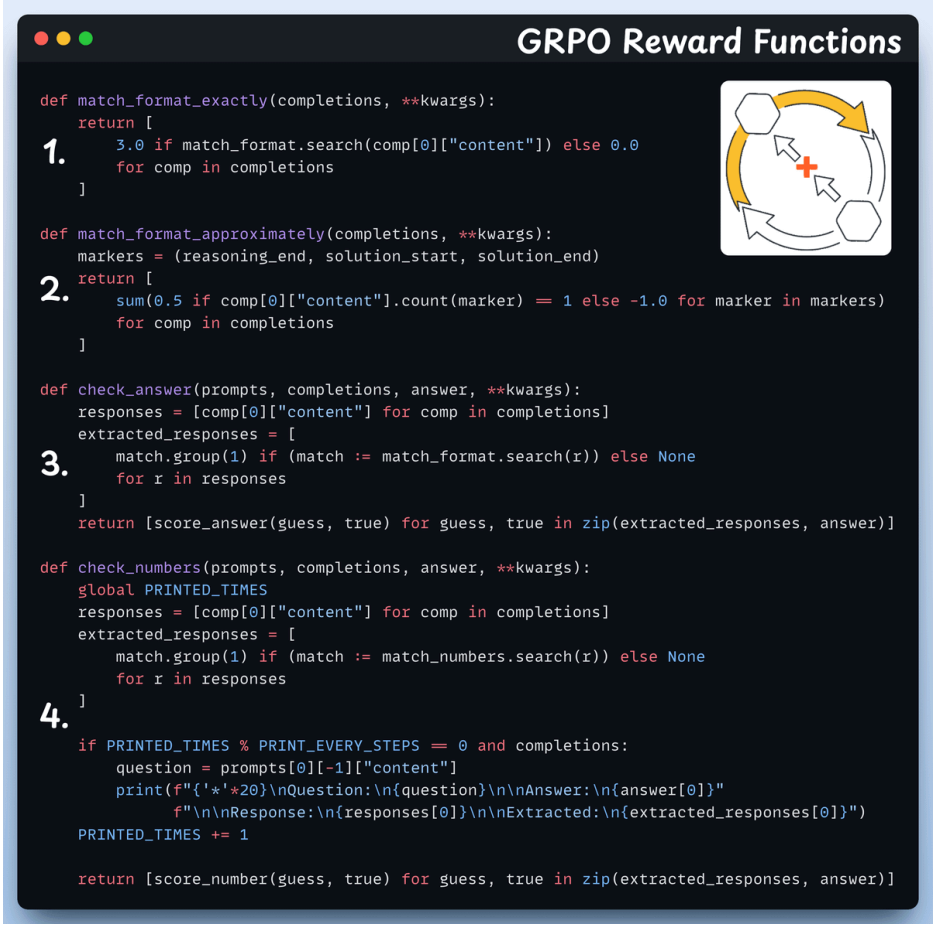
Each sample includes:

- 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 🤗

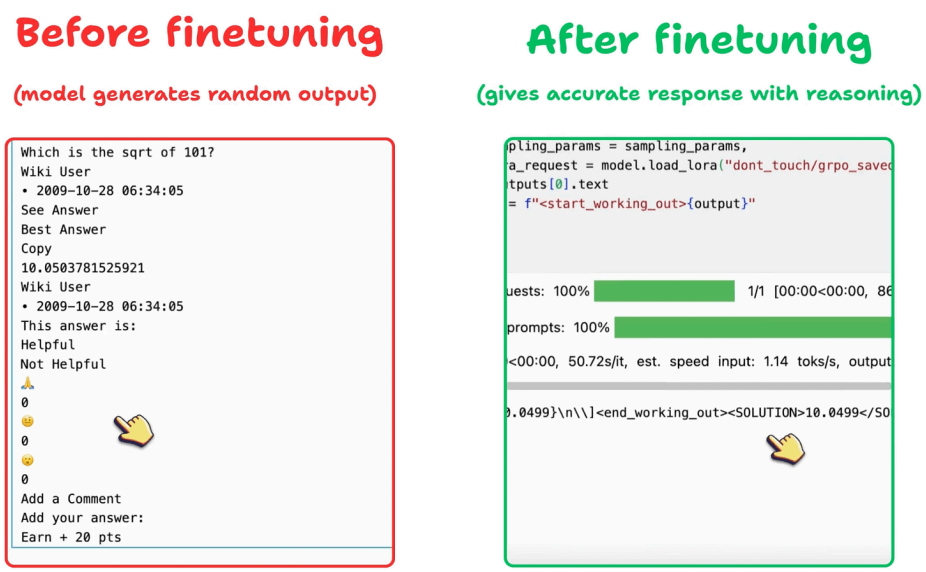
```python
from trl import GRPOTrainer

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

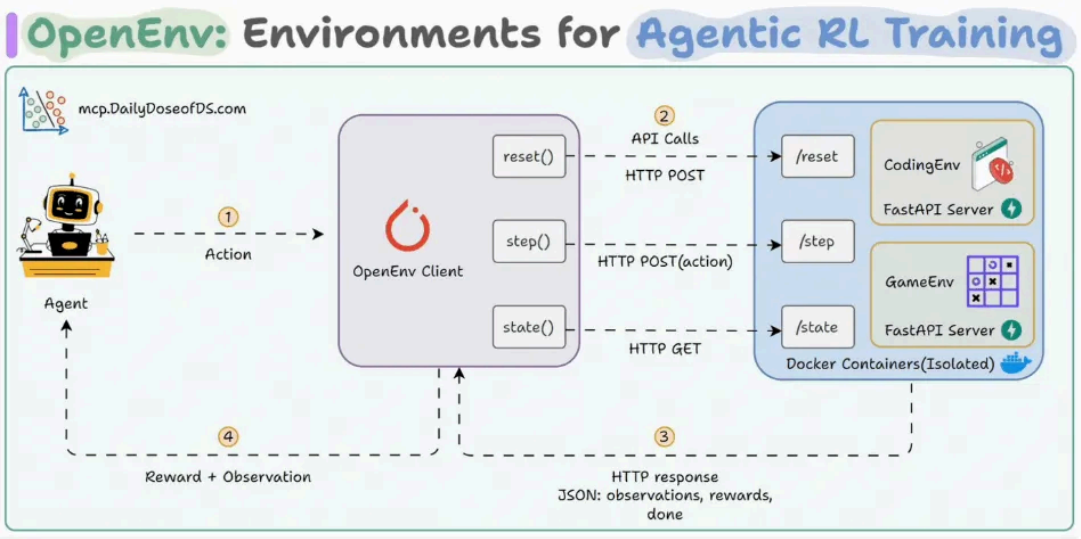
PyTorch OpenEnv is designed to address this lack of standardization. The framework provides a common interface for reinforcement learning environments, inspired by Gymnasium but implemented as a containerized, service-based system.

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.

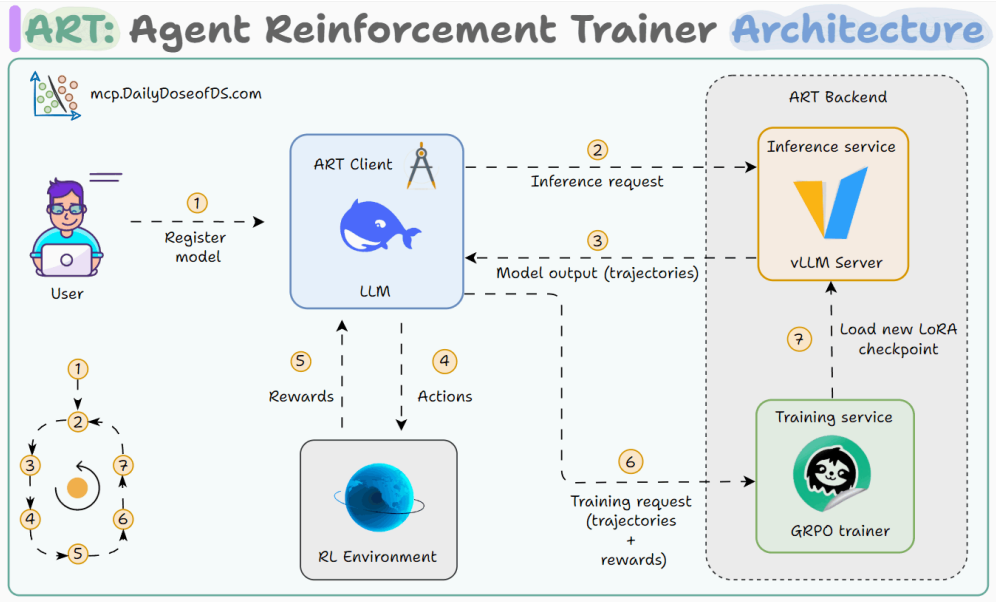
Agent Reinforcement Trainer(ART)

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:

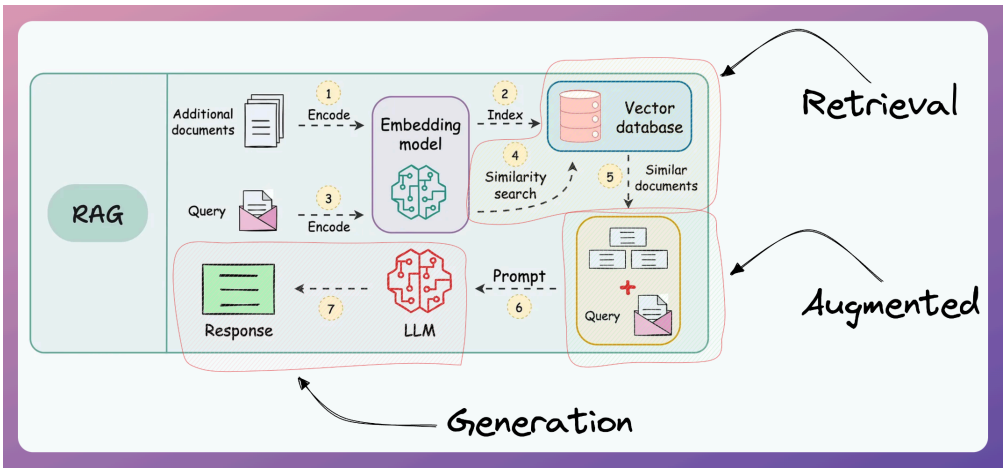
- Prompt Engineering - which steers the model at inference time
- Fine-tuning - which adjusts its internal parameters.

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.

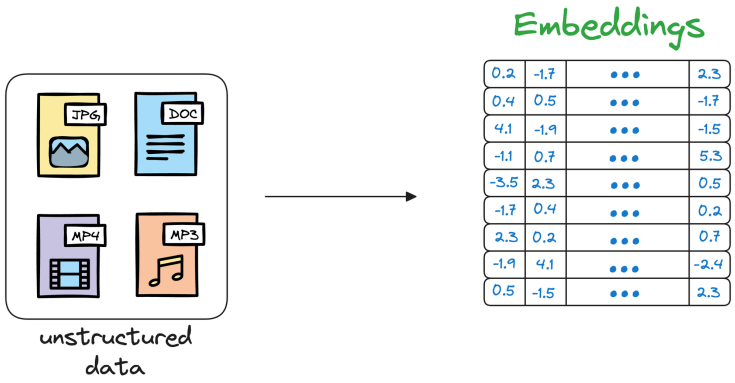
With RAG, the language model can use the retrieved information (which is expected to be reliable) from the vector database to ensure that its responses are grounded in real-world knowledge and context, reducing the likelihood of hallucinations.

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?

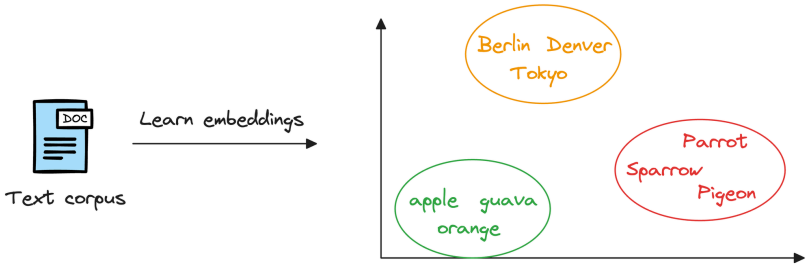
Simply put, a vector database stores unstructured data (text, images, audio, video, etc.) in the form of vector embeddings.



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).

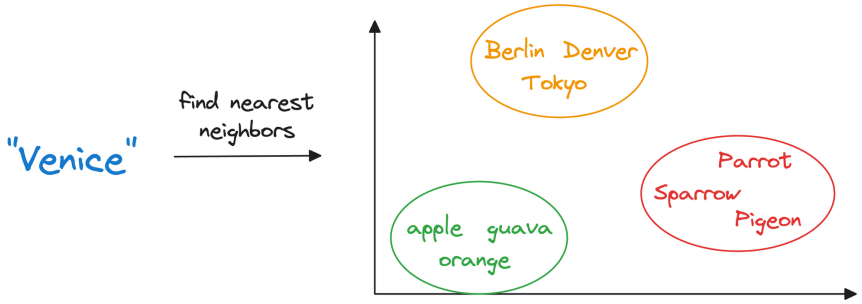
This numerical vector is called an embedding, and the model is trained in such a way that these vectors capture the essential features and characteristics of the underlying data.

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

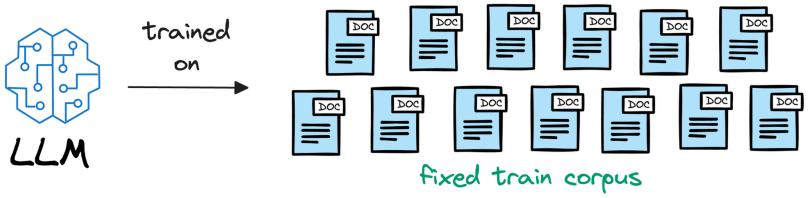
At this point, one interesting thing to learn is how exactly LLMs take advantage of vector databases.

The biggest confusion that people typically face is:

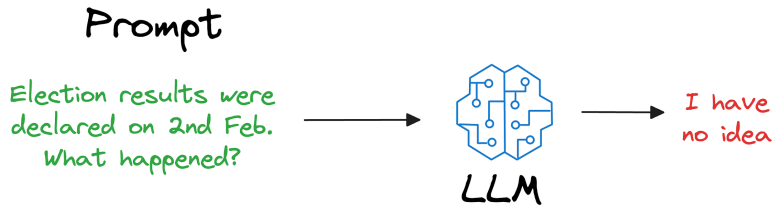
Once we have trained our LLM, it will have some model weights for text generation. Where do vector databases fit in here?

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.



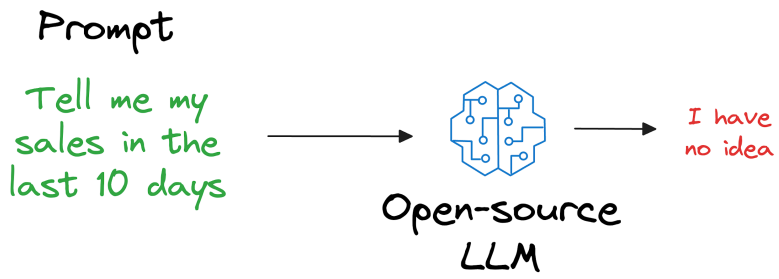
For instance, if the model was deployed after considering the data until 31st Jan 2024, and we use it, say, a week after training, it will have no clue about what happened in those days.



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.

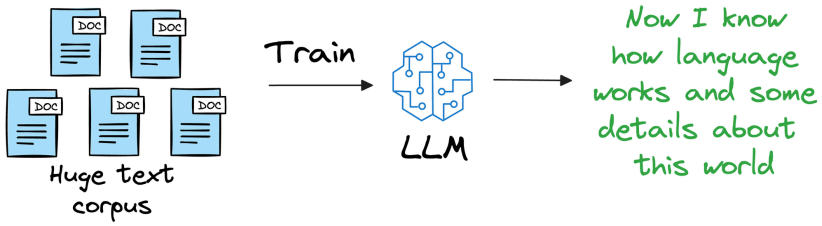


But if you think about it, is it really our objective to train an LLM to know every single thing in the world?

Not at all!

That's not our objective.

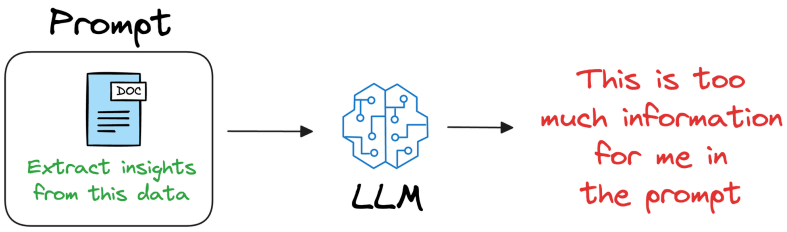
Instead, it is more about helping the LLM learn the overall structure of the language, and how to understand and generate it.



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.

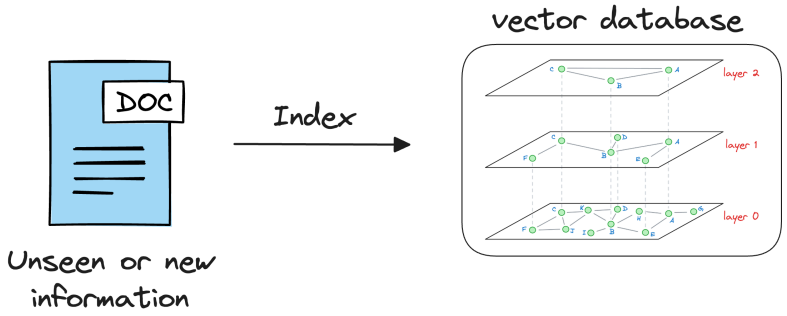


But since LLMs usually have a limit on the context window (number of words/tokens they can accept), the additional information can exceed that limit.

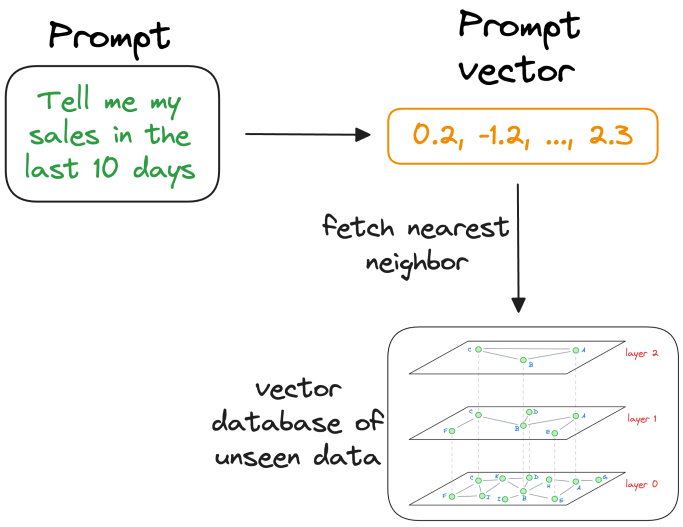
Vector databases solve this problem.

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.

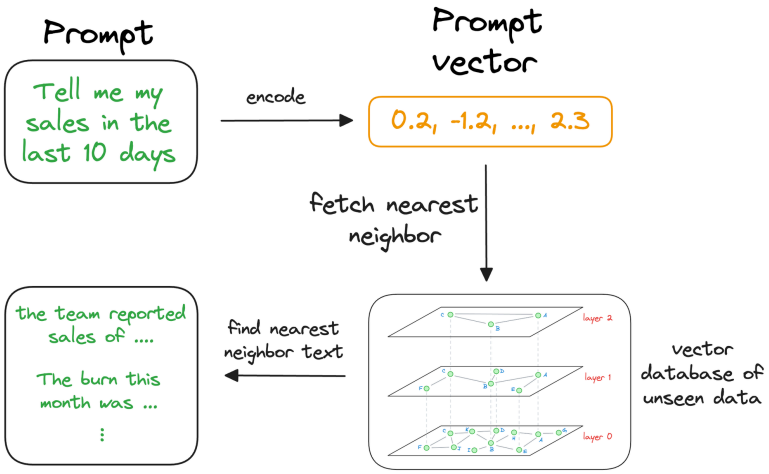
Thus, we can maintain our available information in a vector database by encoding it into vectors using an embedding model.



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.

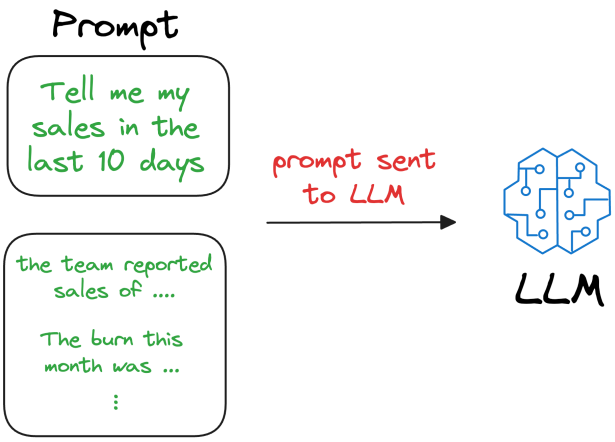


Once the approximate nearest neighbors have been retrieved, we gather the context corresponding to those specific vectors, which were stored at the time of indexing the data in the vector database (this raw data is stored as payload, which we will learn during implementation).



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.



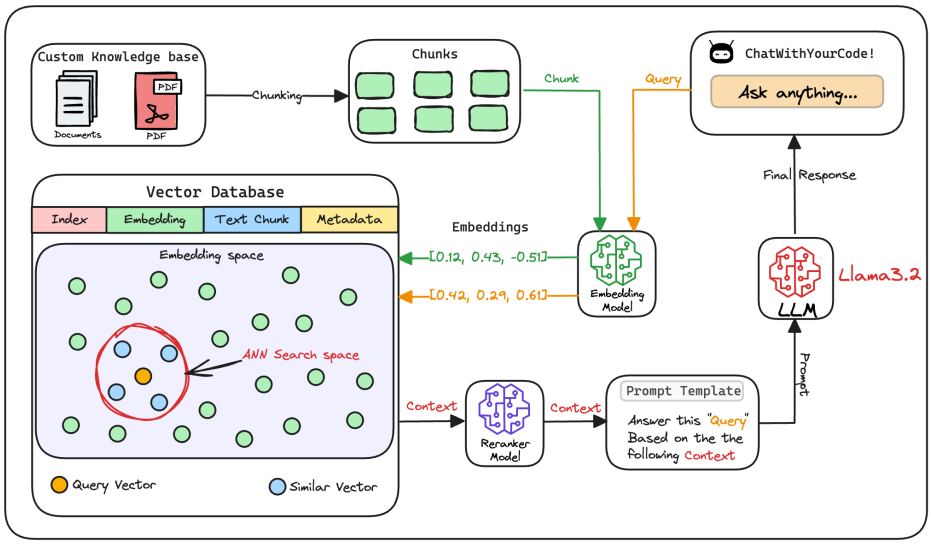
Consequently, the LLM can easily incorporate this info while generating text because it now has the relevant details available in the prompt.

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

Workflow of a RAG system

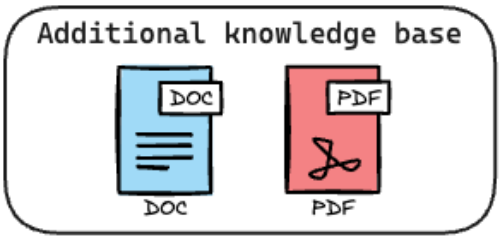
To build a RAG system, it's crucial to understand the foundational components that go into it and how they interact. Thus, in this section, let's explore each element in detail.

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



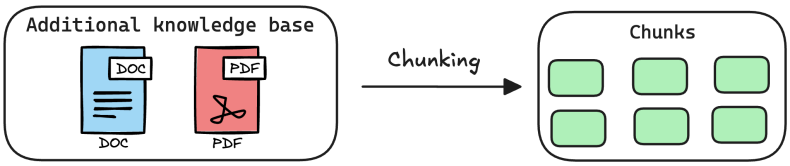
Let's break it down step by step.

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

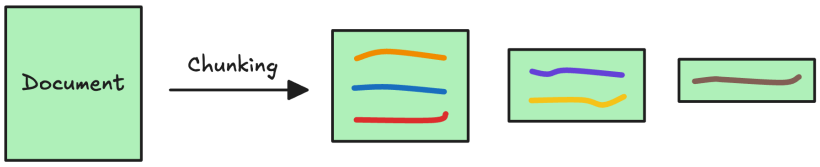


#1) Create chunks

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



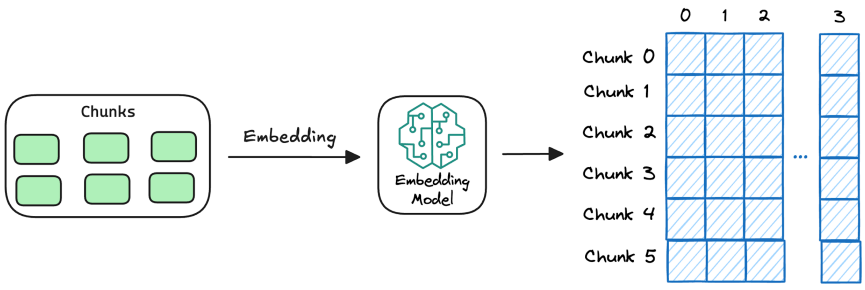
We do this because the additional document(s) can be pretty large. Thus, it is important to ensure that the text fits the input size of the embedding model.



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

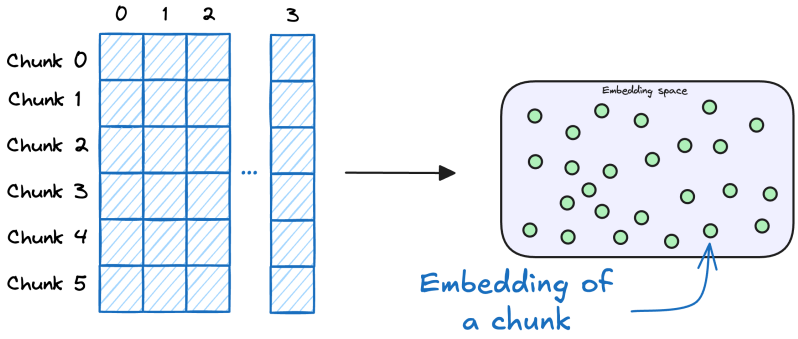
After chunking, we embed the chunks using an embedding model.



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.

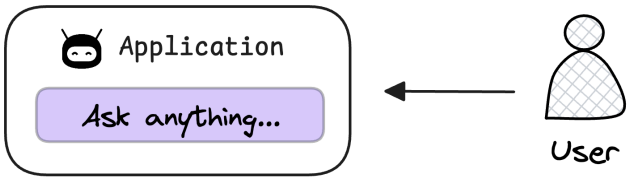
A vector database also stores the metadata and original content along with the vector embeddings.

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.

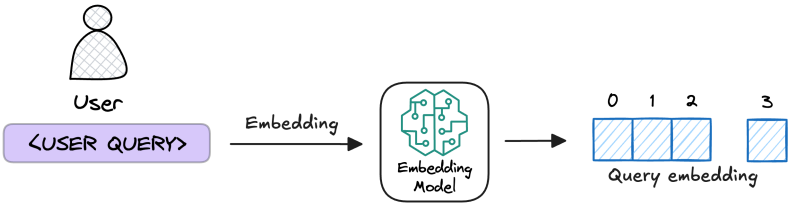
#4) User input query

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



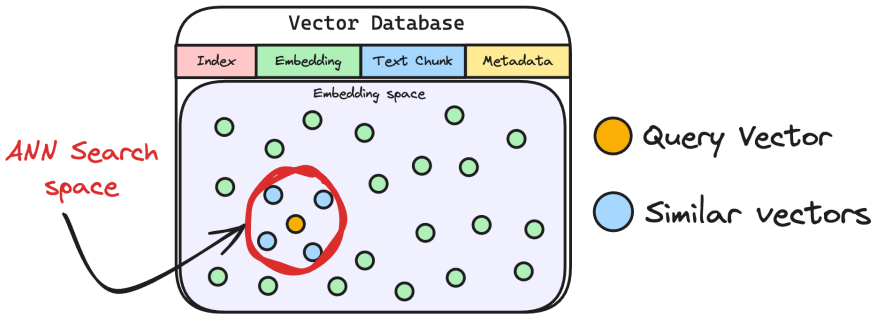
#5) Embed the query

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

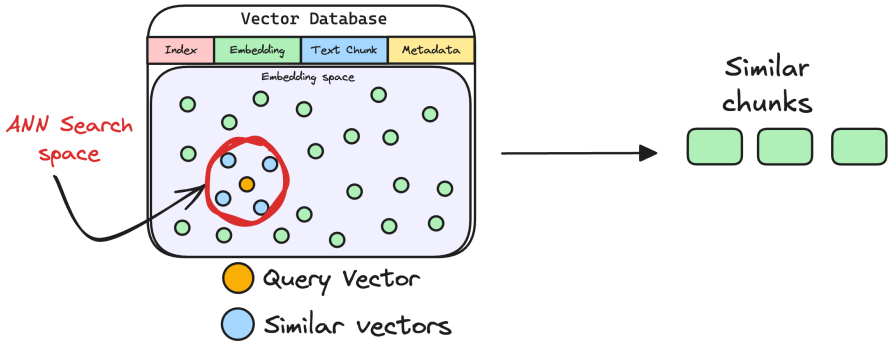


#6) Retrieve similar chunks

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).

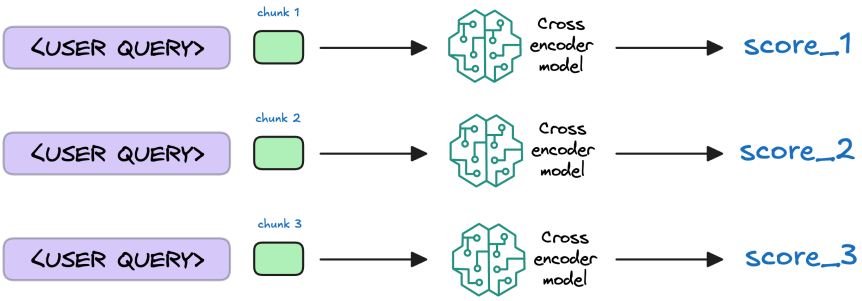


It is expected that these retrieved documents contain information related to the query, providing a basis for the final response generation.

#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.

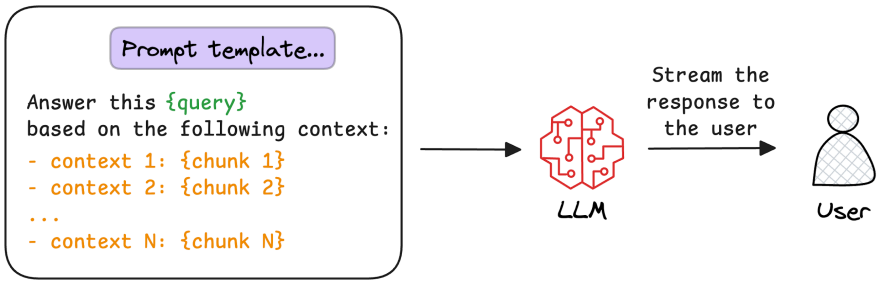
#8) Generate the final response

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:

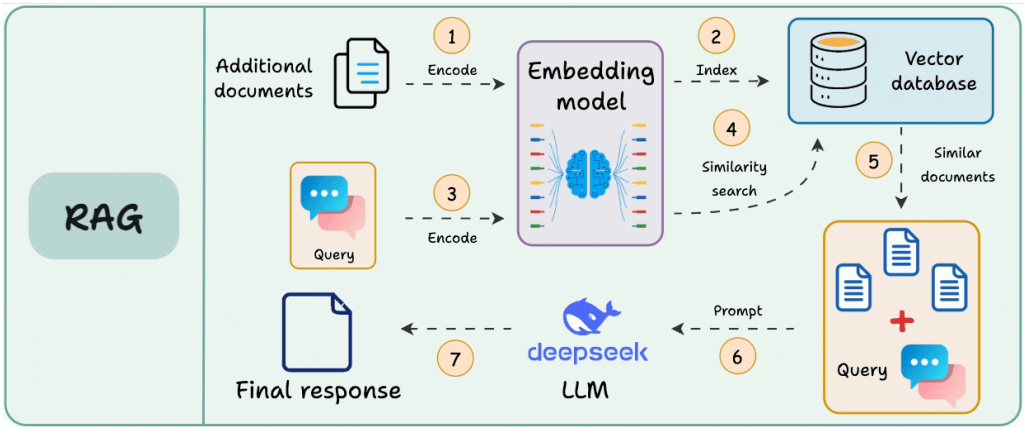


The LLM leverages the context provided by the chunks to generate a coherent and contextually relevant answer that directly addresses the user's query.

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

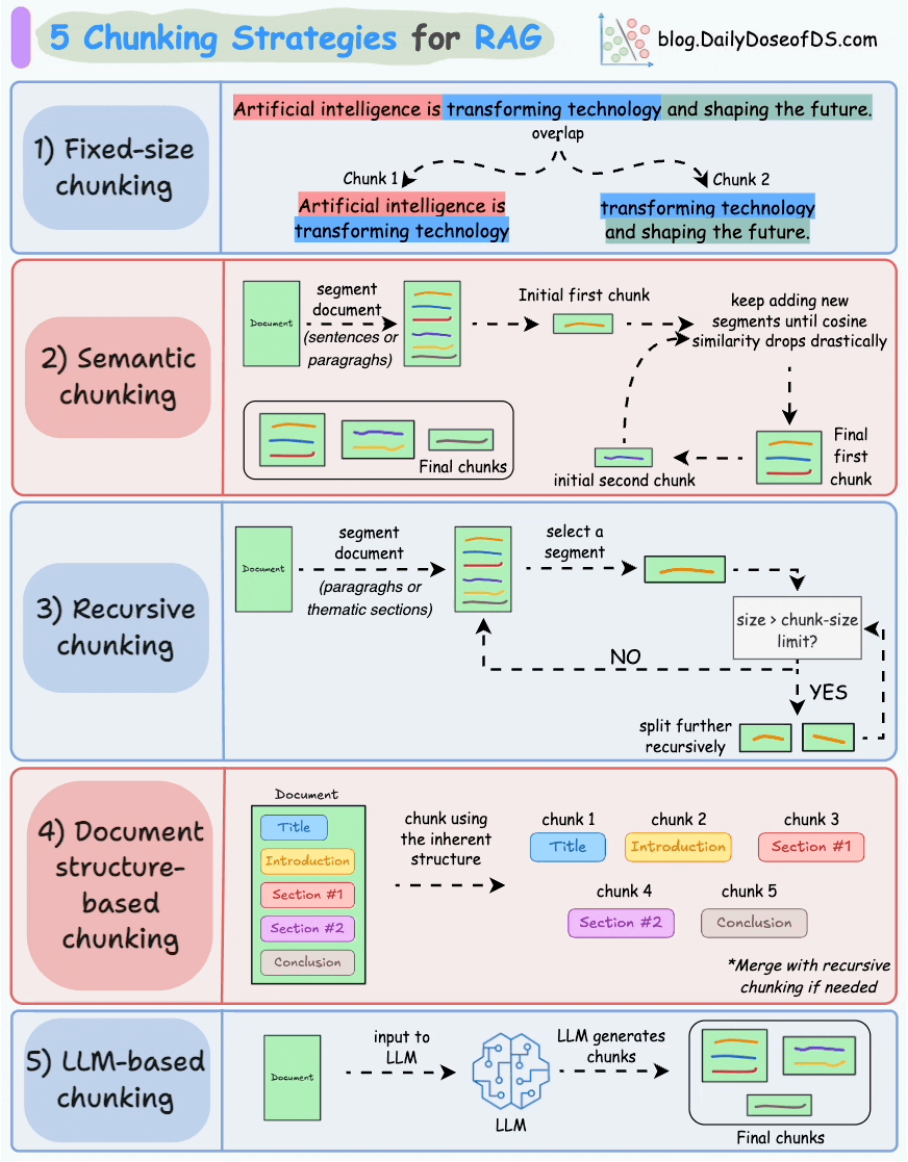
Here's the typical workflow of RAG:



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.

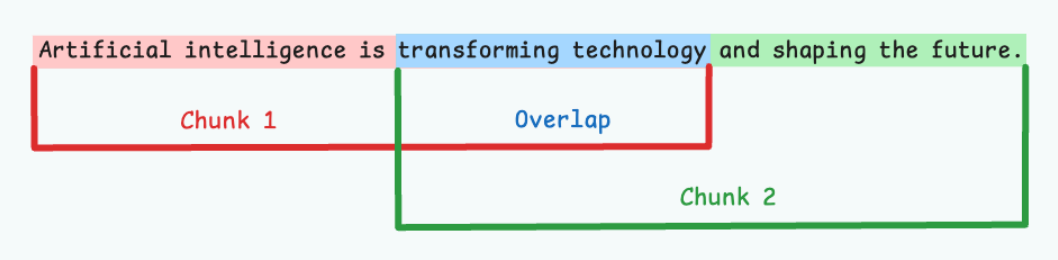
Here are five chunking strategies for RAG:



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.

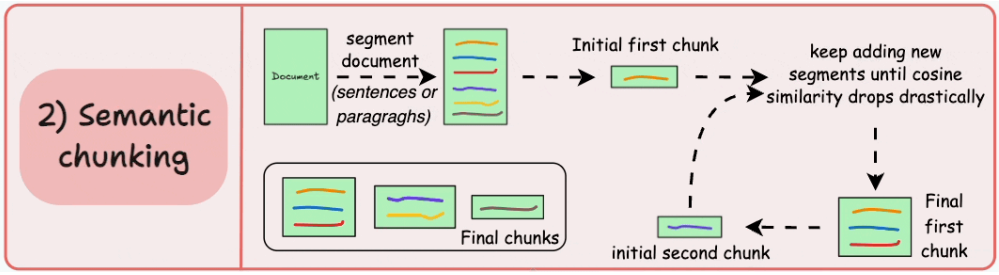


Since a direct split can disrupt the semantic flow, it is recommended to maintain some overlap between two consecutive chunks (the blue part above).

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:

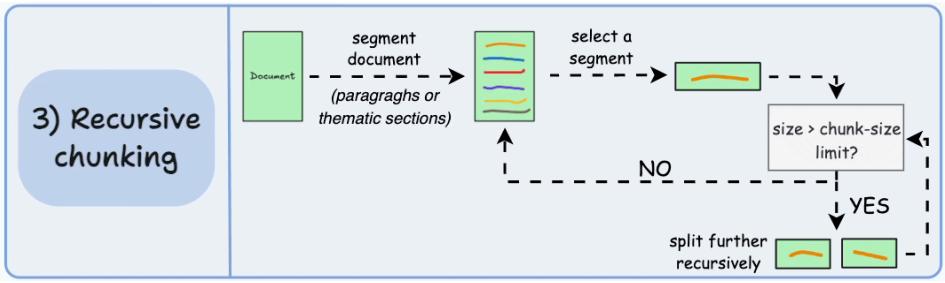
```
Artificial intelligence is transforming industries by automating
processes, enhancing decision-making, and providing insights through
data analysis. Machine learning, a subset of AI, enables systems to learn
and improve from experience without explicit programming. Deep learning,
a branch of machine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

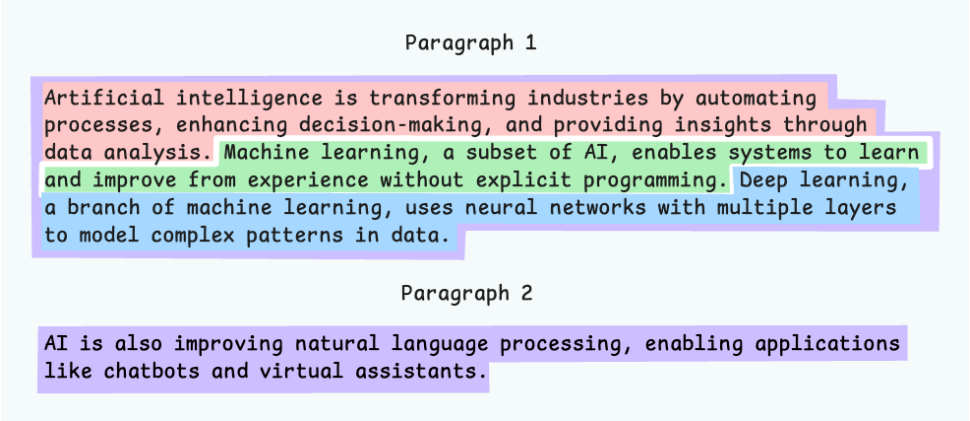
3) Recursive chunking



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:

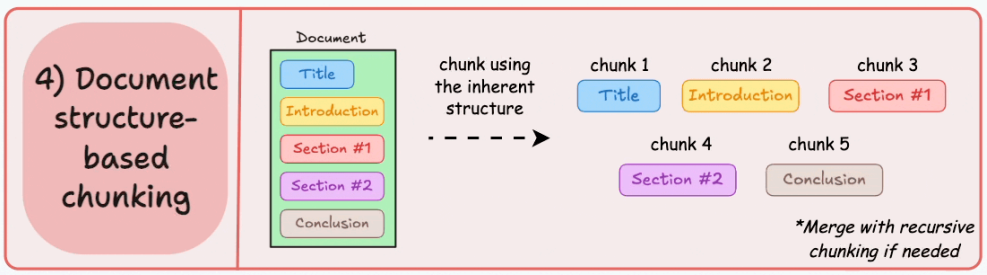
First, we define two chunks (the two paragraphs in purple).

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.

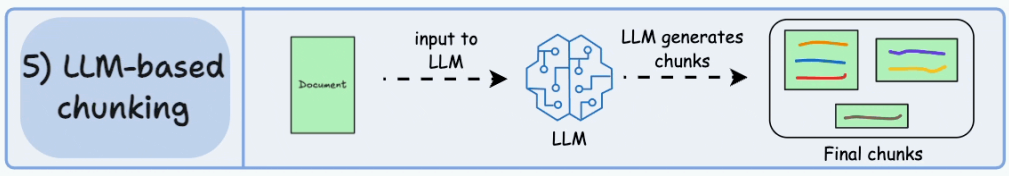
Here's what the output could look like:

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.

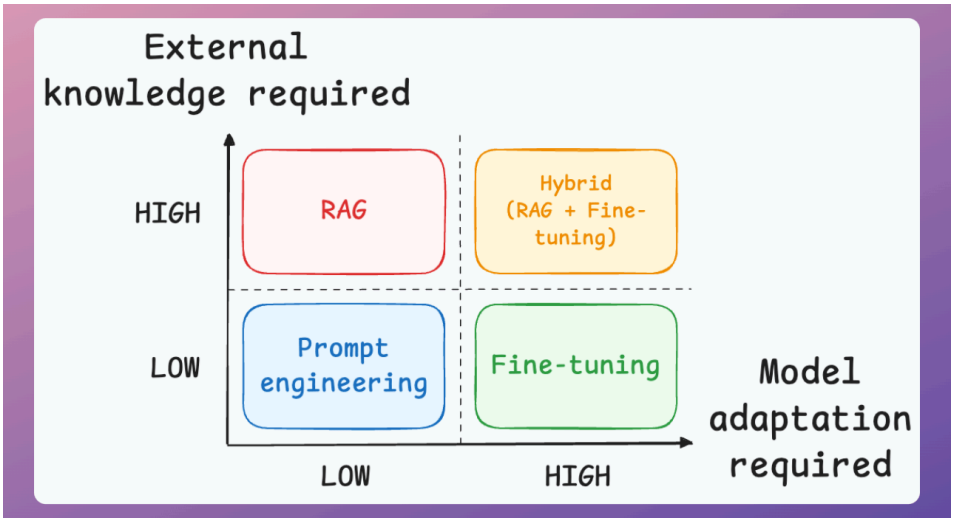
The choice will depend on the nature of your content, the capabilities of the embedding model, computational resources, etc.

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:

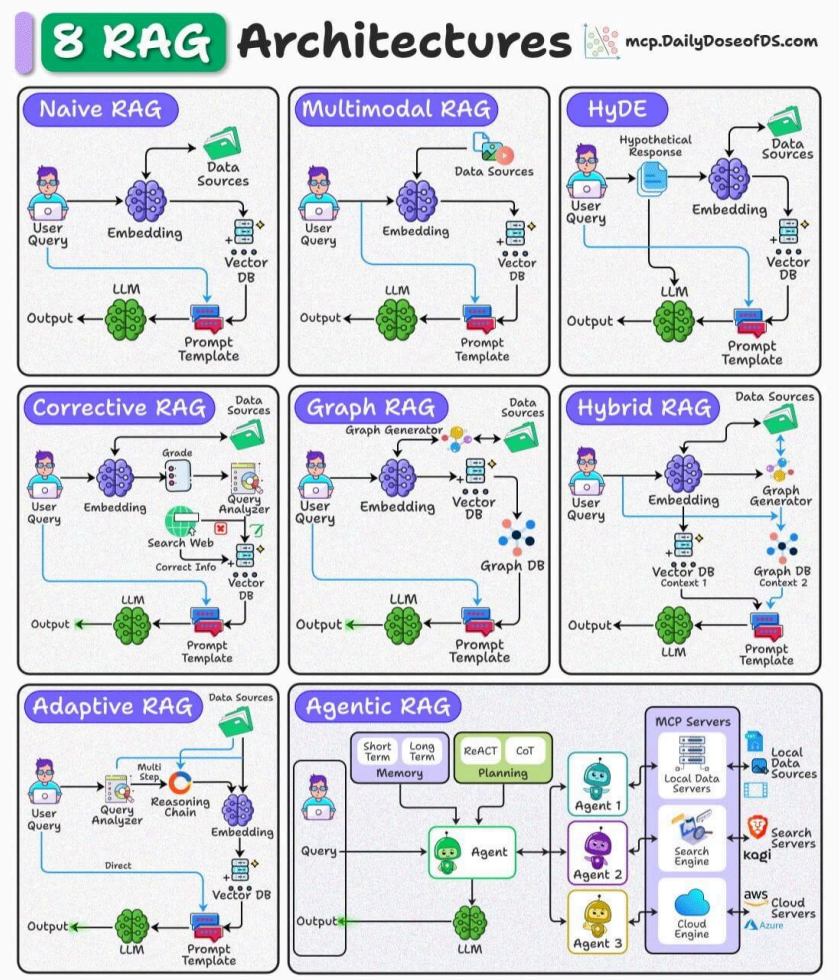
- Use RAGs to generate outputs based on a custom knowledge base if the vocabulary & writing style of the LLM remains the same.
- Use fine-tuning to change the structure (behaviour) of the model than knowledge.
- Prompt engineering is sufficient if you don't have a custom knowledge base and don't want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

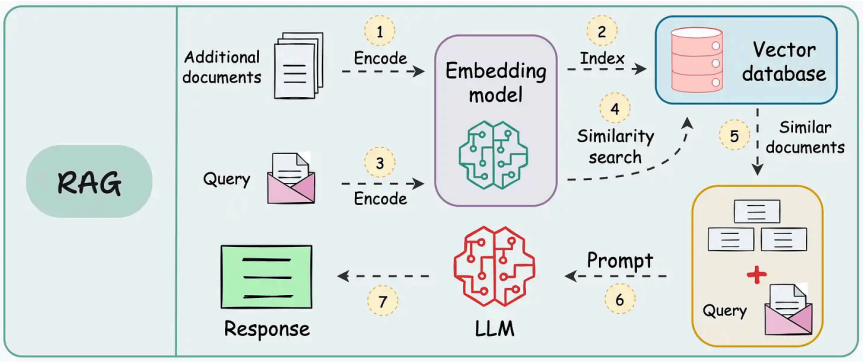
8) Agentic RAG

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.

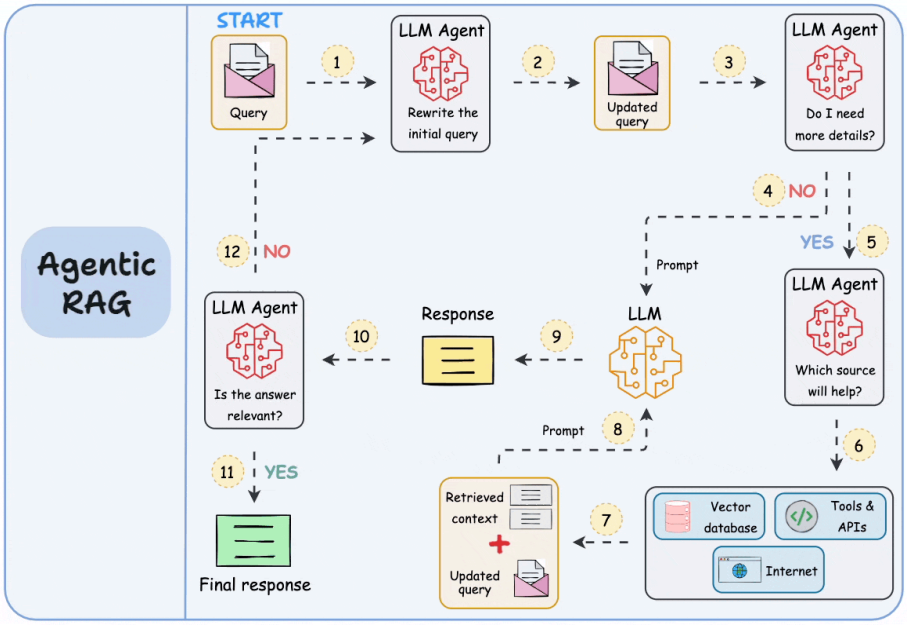
RAG systems may provide relevant context but don't reason through complex queries. If a query requires multiple retrieval steps, traditional RAG falls short.

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.

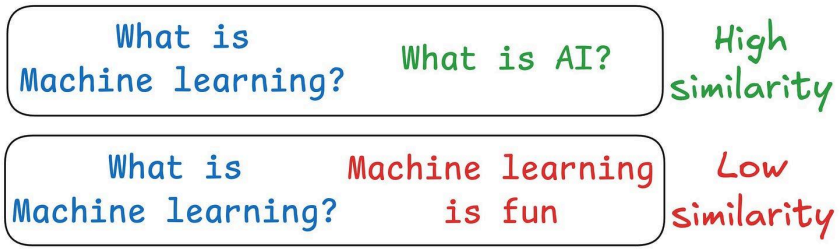
This makes the RAG much more robust since, at every step, agentic behavior ensures that individual outcomes are aligned with the final goal.

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

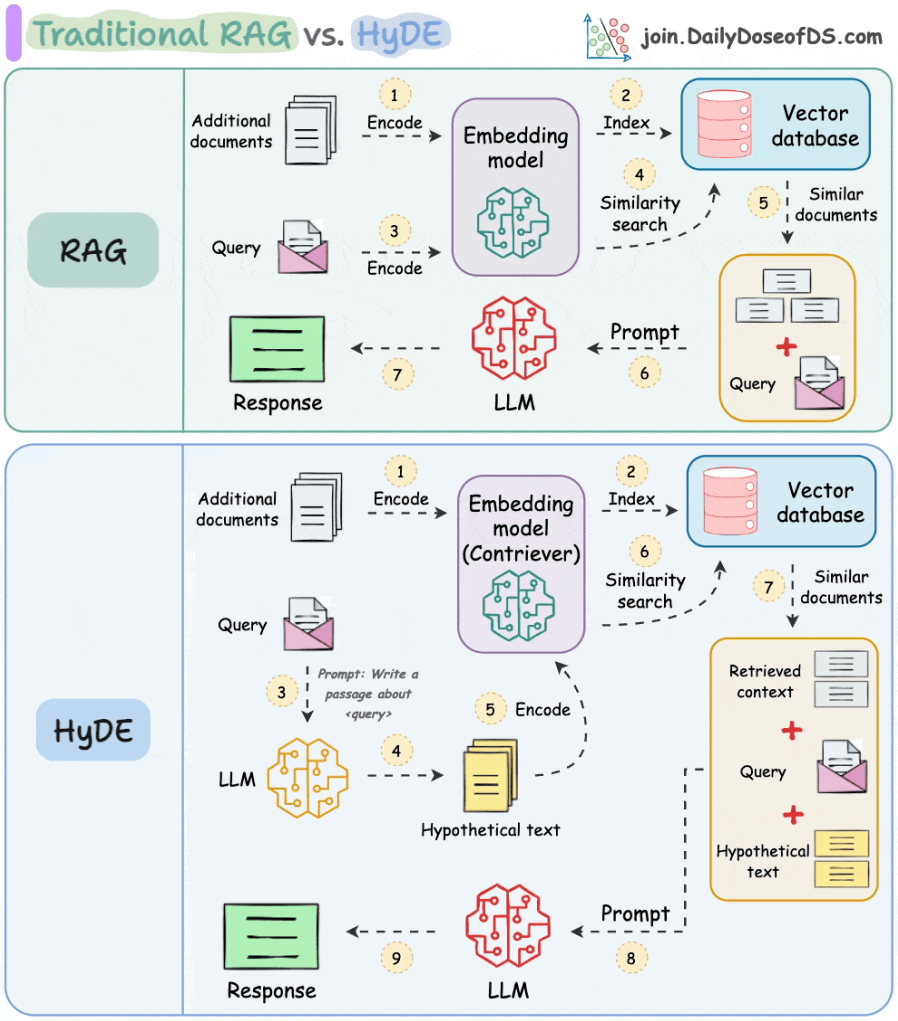
Another critical problem with the traditional RAG system is that questions are not semantically similar to their answers.



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.

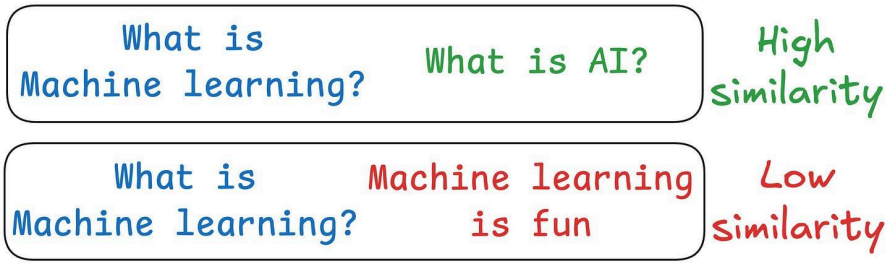
HyDE solves this.

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:

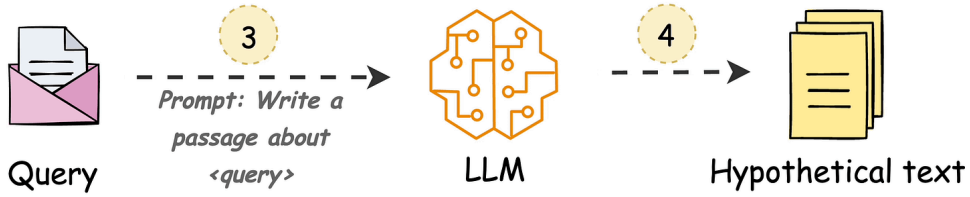
Use an LLM to generate a hypothetical answer H for the query Q (this answer does not have to be entirely correct).

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.

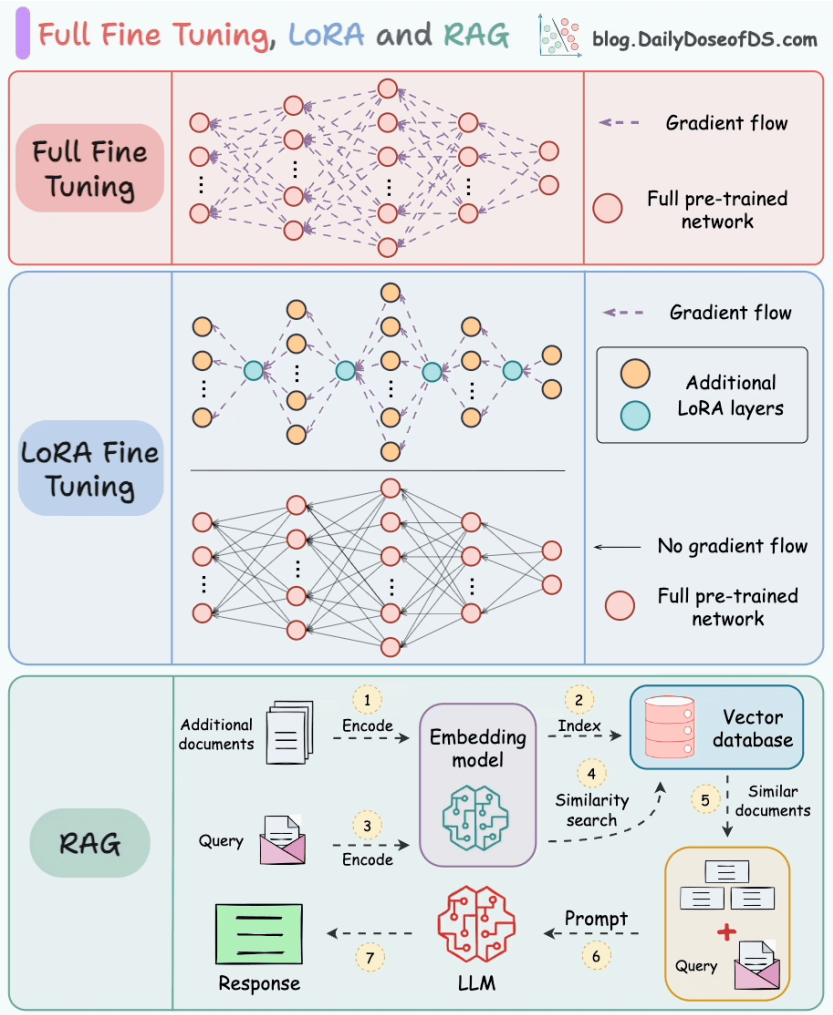
More specifically, this model is trained using contrastive learning and it also functions as a near-lossless compressor whose task is to filter out the hallucinated details of the fake document.

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.

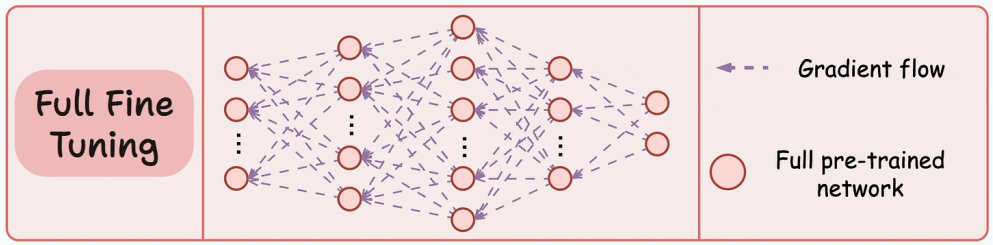
Full-model Fine-tuning vs. LoRA vs. RAG



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.

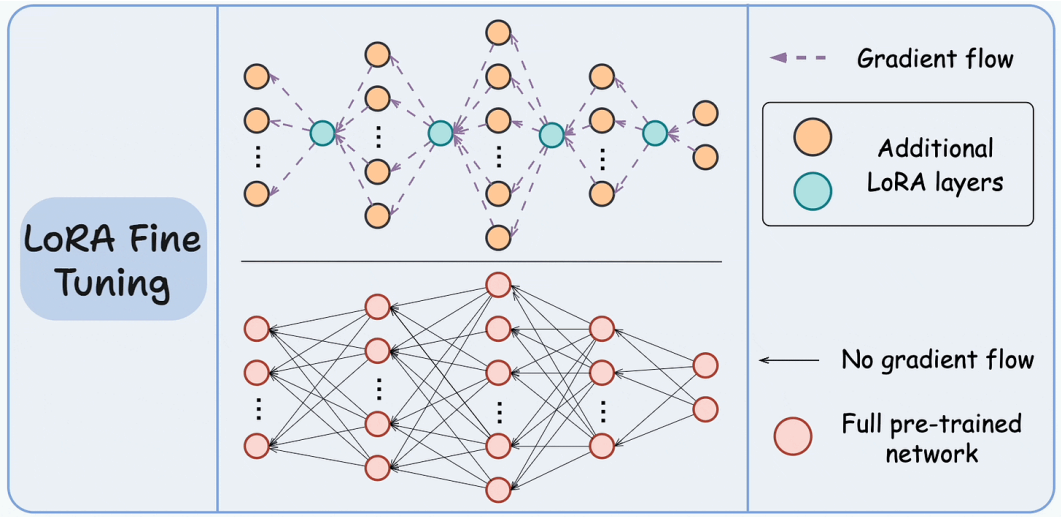
The cost involved in maintaining all large fine-tuned models.

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.

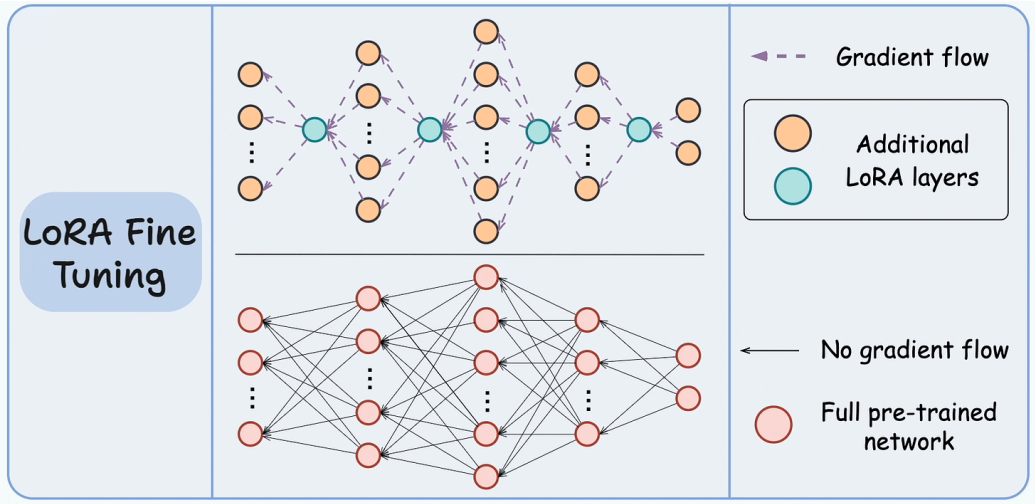
Looking at the above visual, you might think:

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:

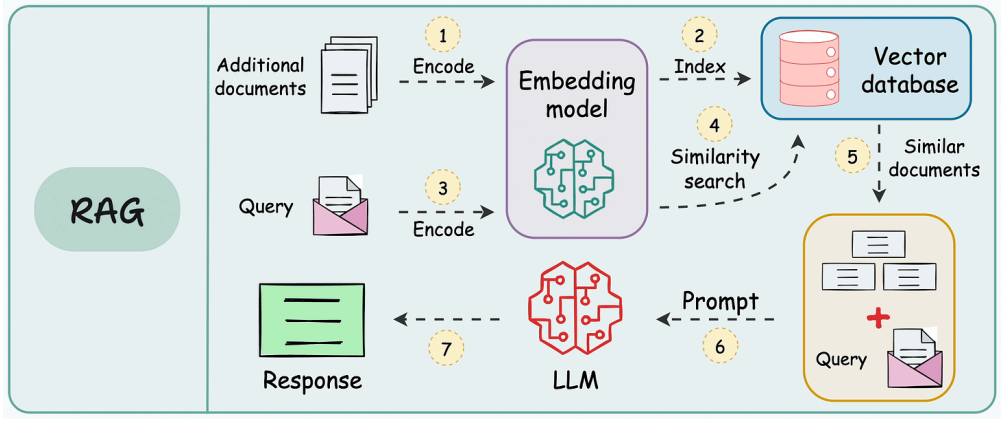


Looking at the above visual, it is pretty clear that the LoRA network has relatively very few connections.

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

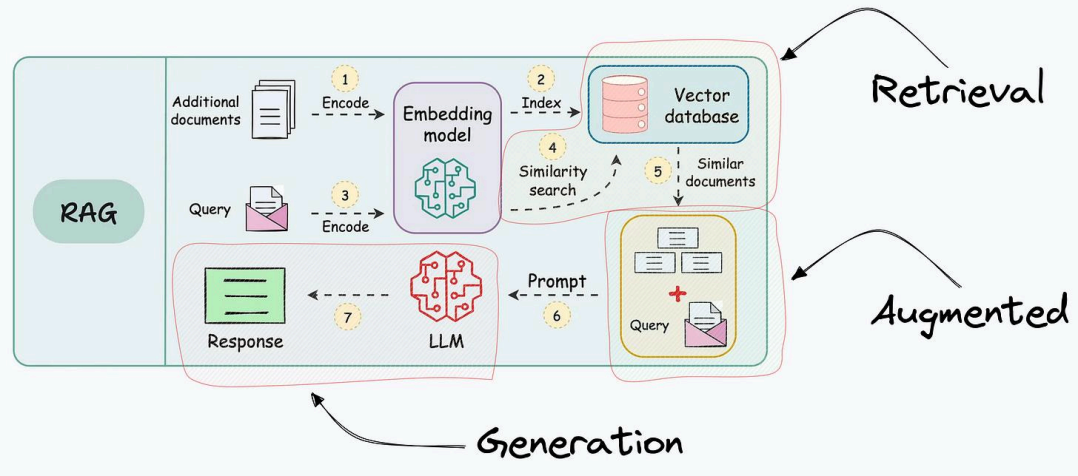
embeddings into the vector database. There's no need to repeat this again for the entire data)

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:

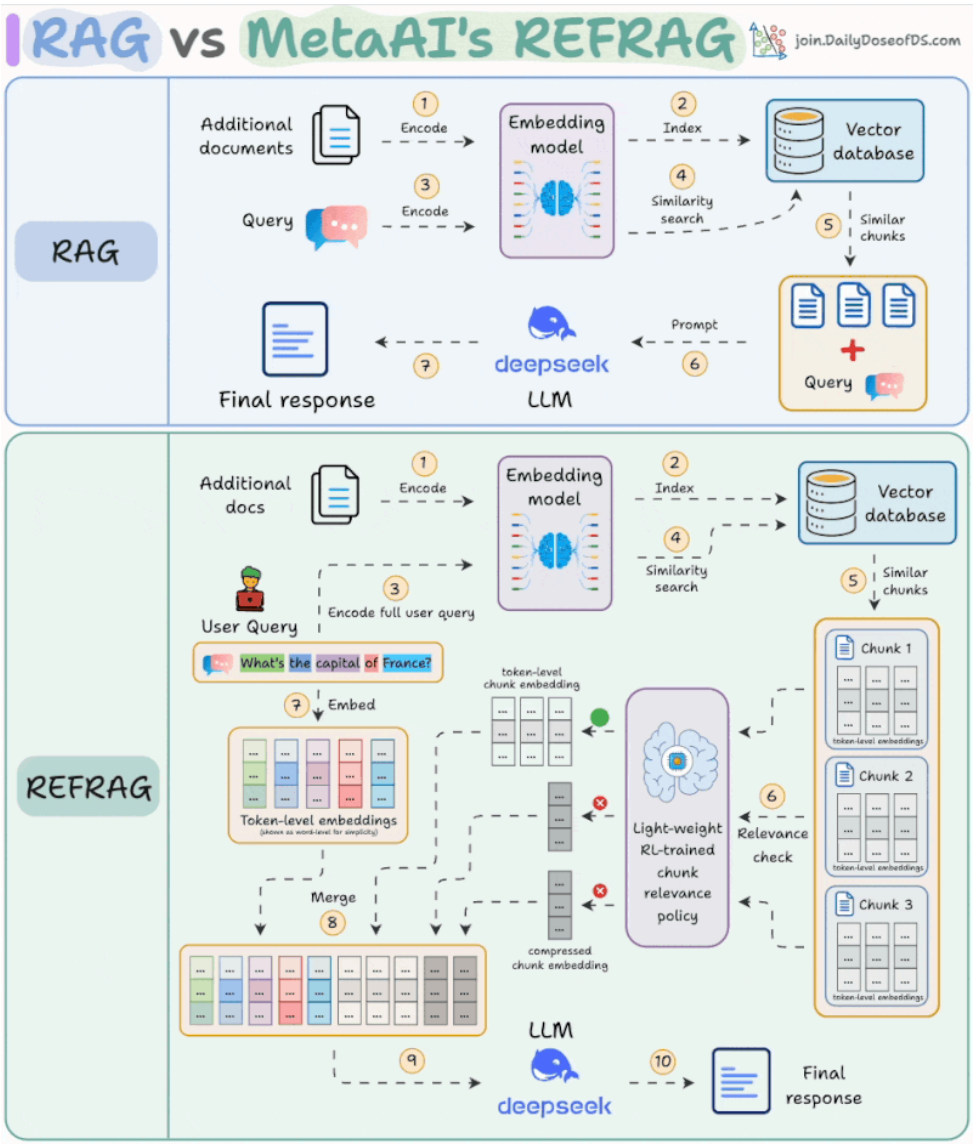
- You encode it into a vector.
- Fetch similar chunks from vector DB.
- Dump the retrieved context into the LLM.

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

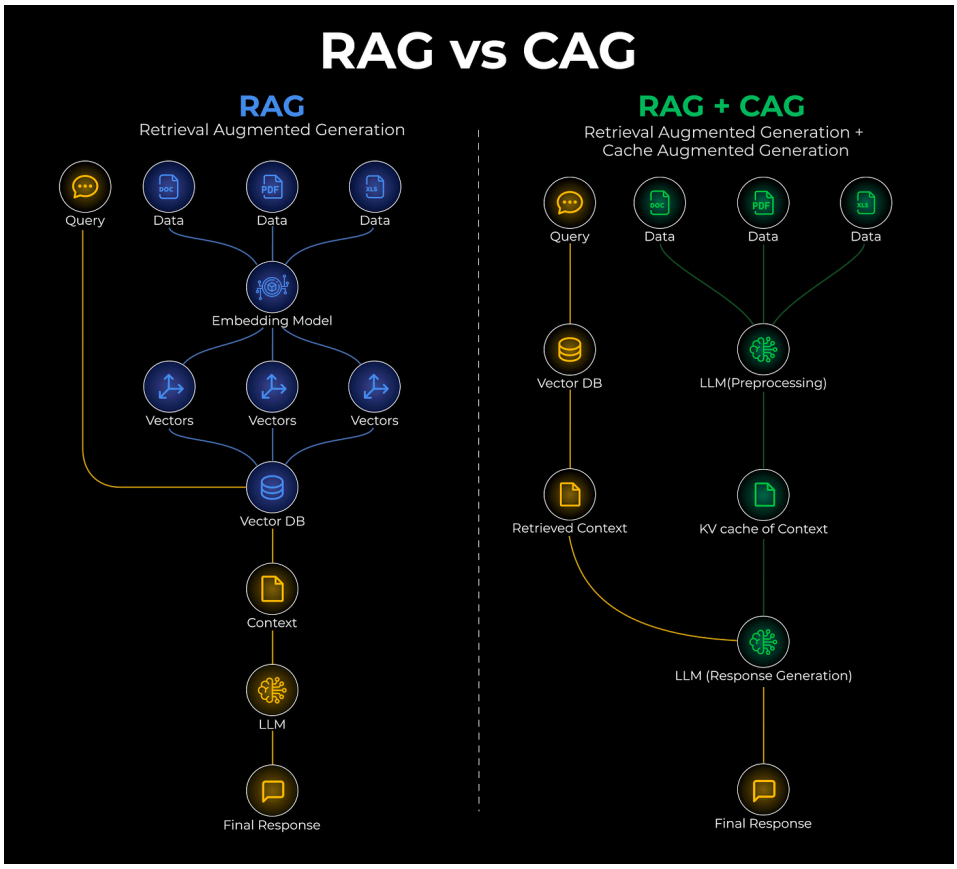
RAG changed how we build knowledge-grounded systems, but it still has a weakness.

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.

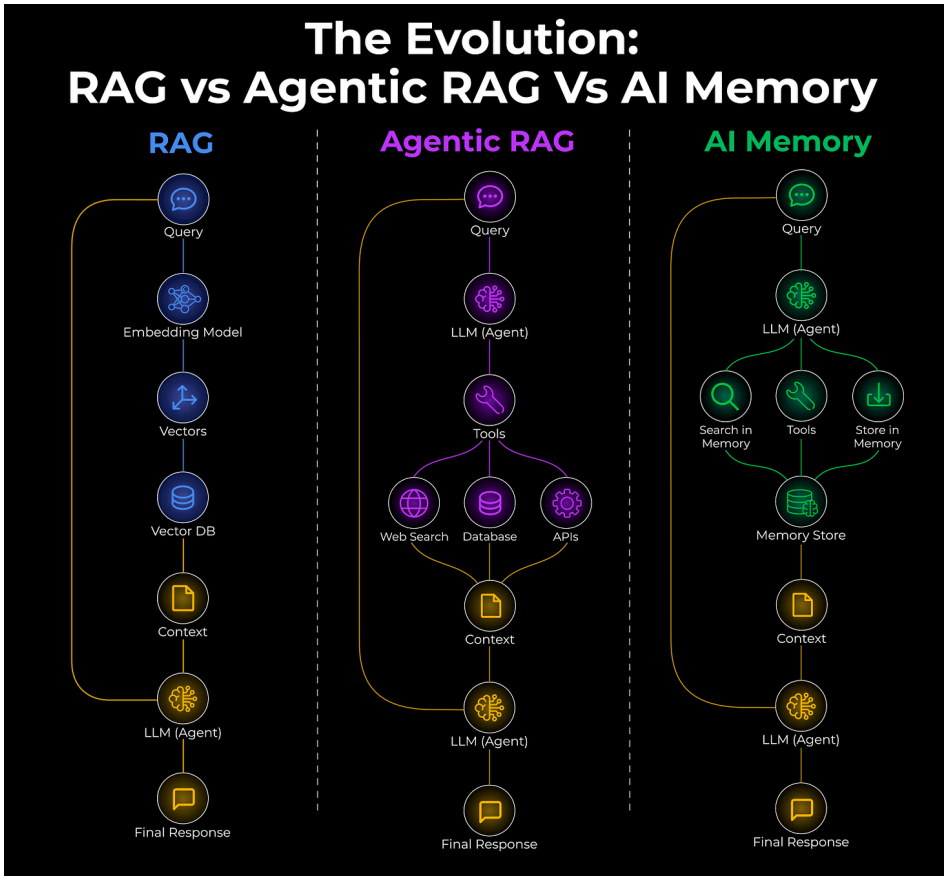
You can also see this in practice above.

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

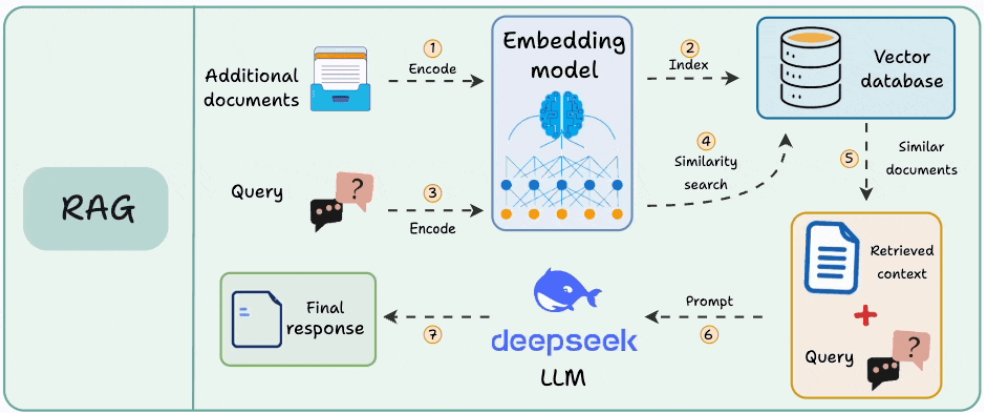
What is Context Engineering?

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.

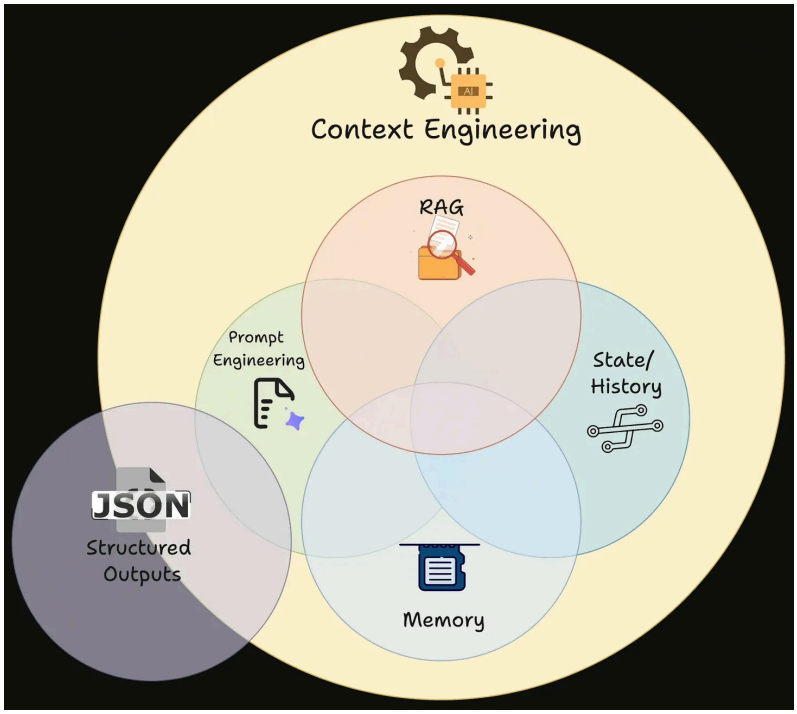


Thus:

- 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.

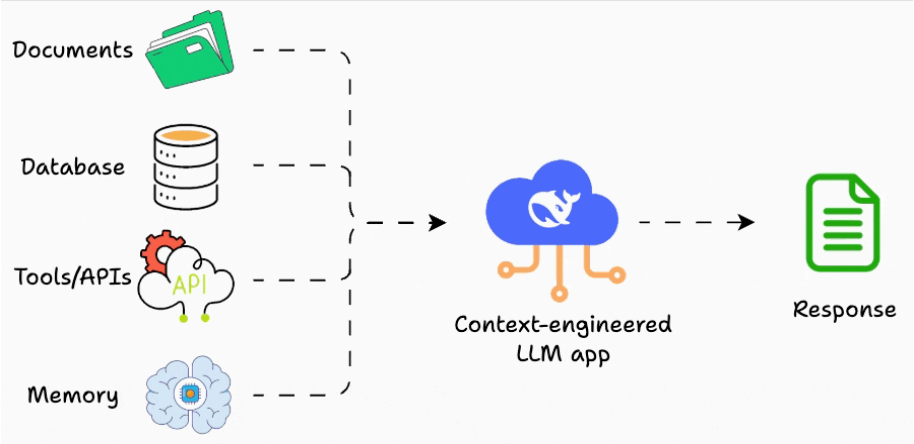
But why was traditional prompt engineering not enough?

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…

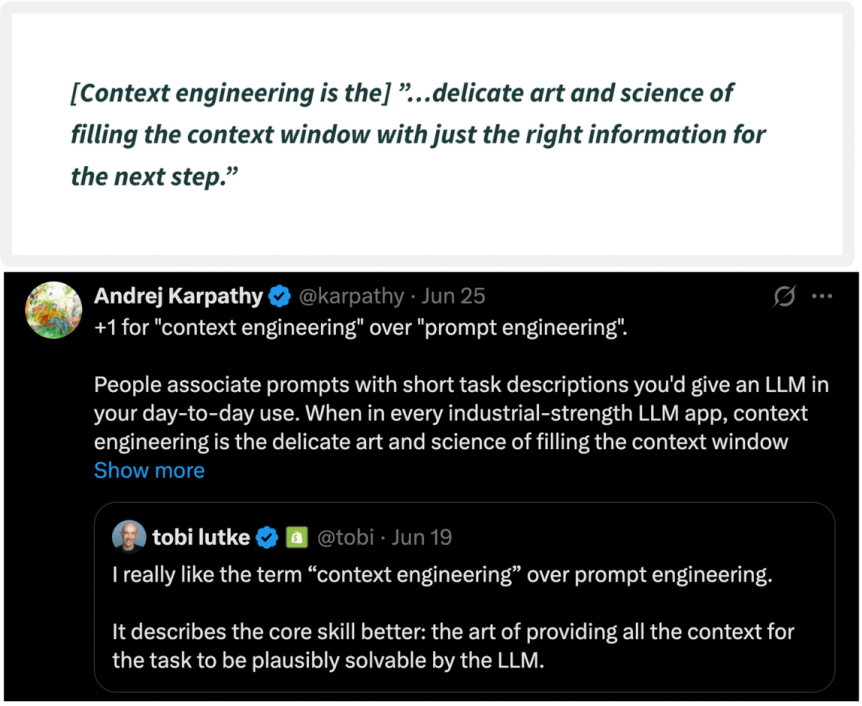
Context engineering is becoming the new core skill since it addresses the real bottleneck, which is not model capability, but setting up an architecture of information.

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.

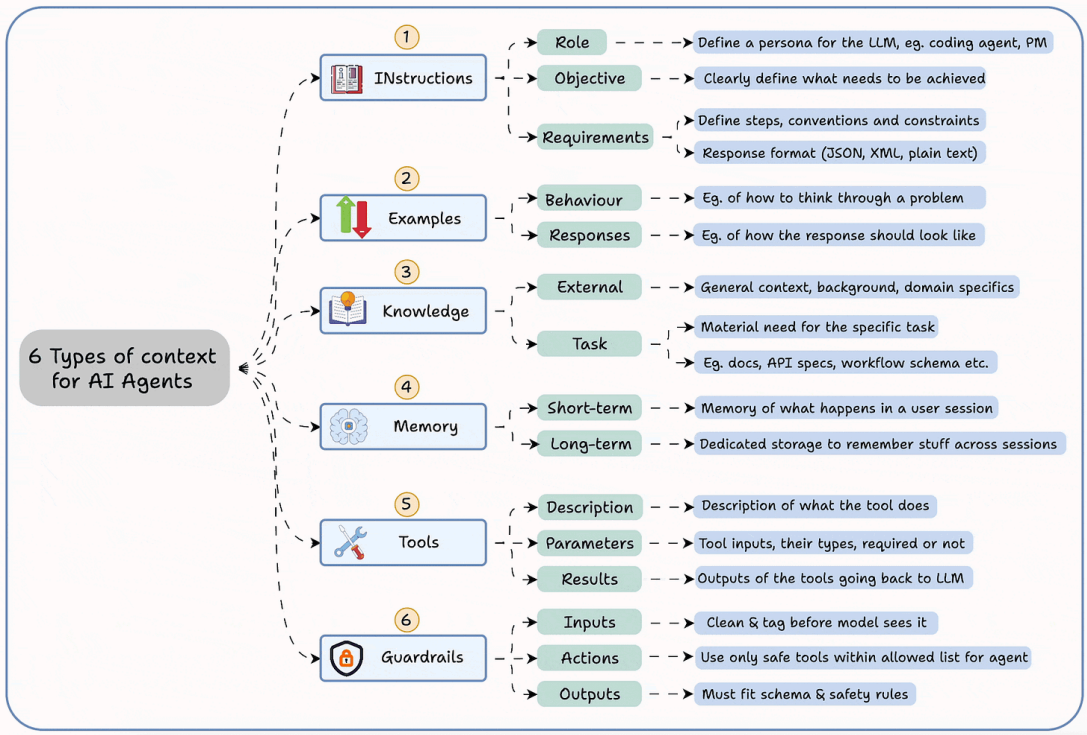
Here's a quote by Andrej Karpathy on context engineering...



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:

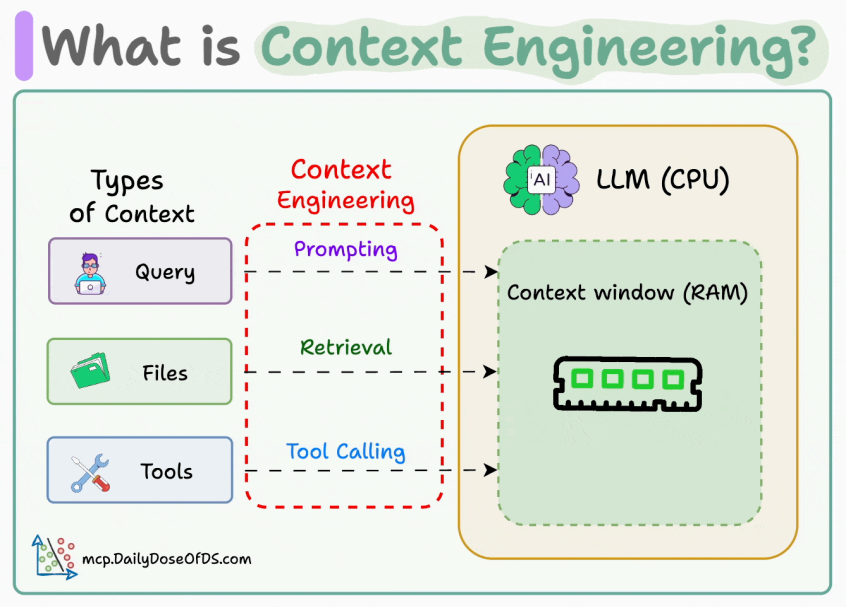


- Instructions
- Examples
- Knowledge
- Memory
- Tools
- Guardrails

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

You must engineer the input (context).

Think of it this way:

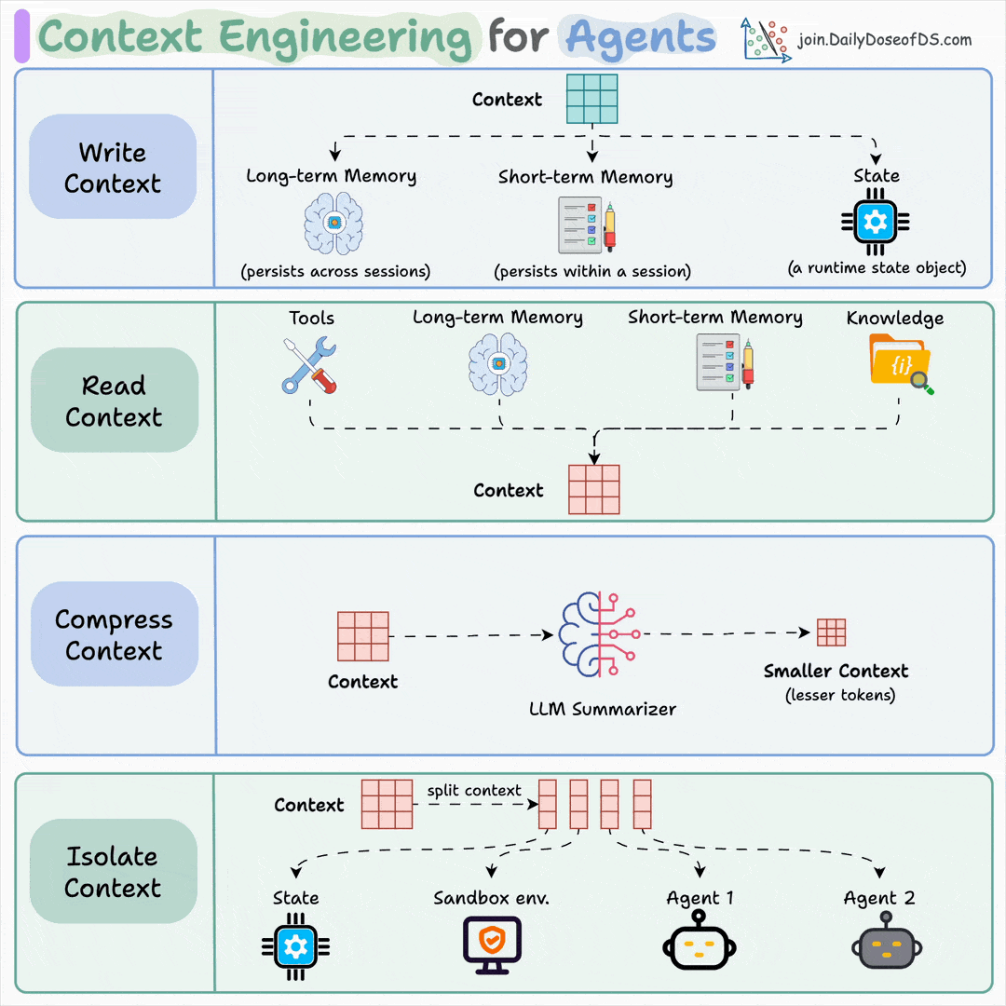


- If LLM is a CPU.
- Then the context window is the RAM.

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:

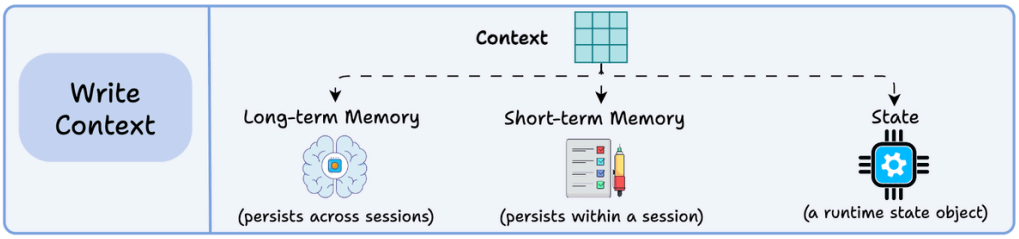


- Writing Context
- Selecting Context
- Compressing Context
- Isolating Context

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.

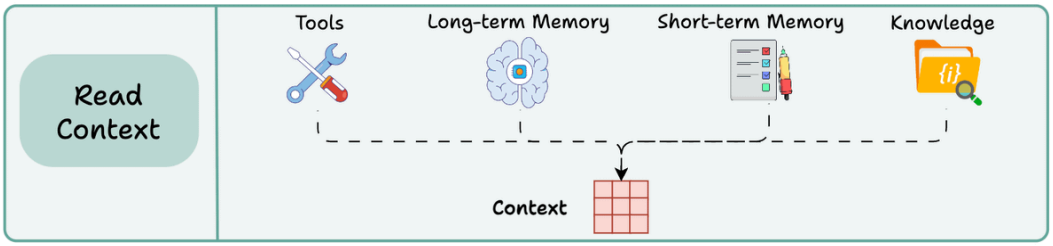


You can do so by writing it to:

- 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.

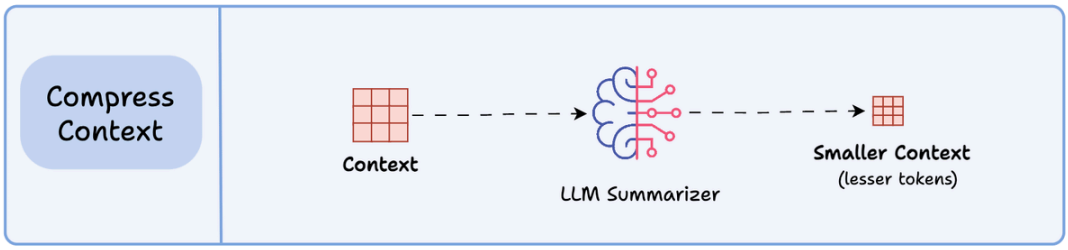


Now this context can be pulled from:

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

3) Compressing context

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

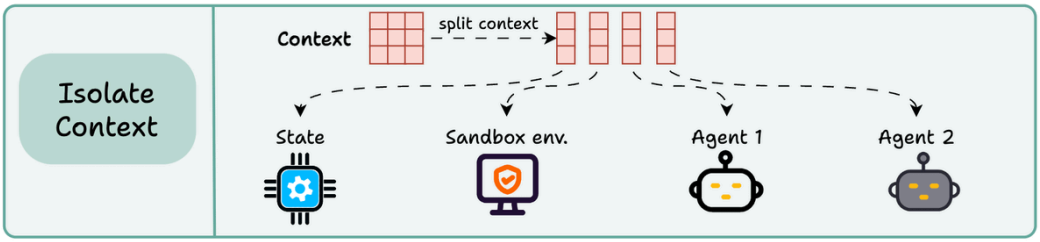


The retrieved context may contain duplicate or redundant information (multi-turn tool calls), leading to extra tokens & increased cost.

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!

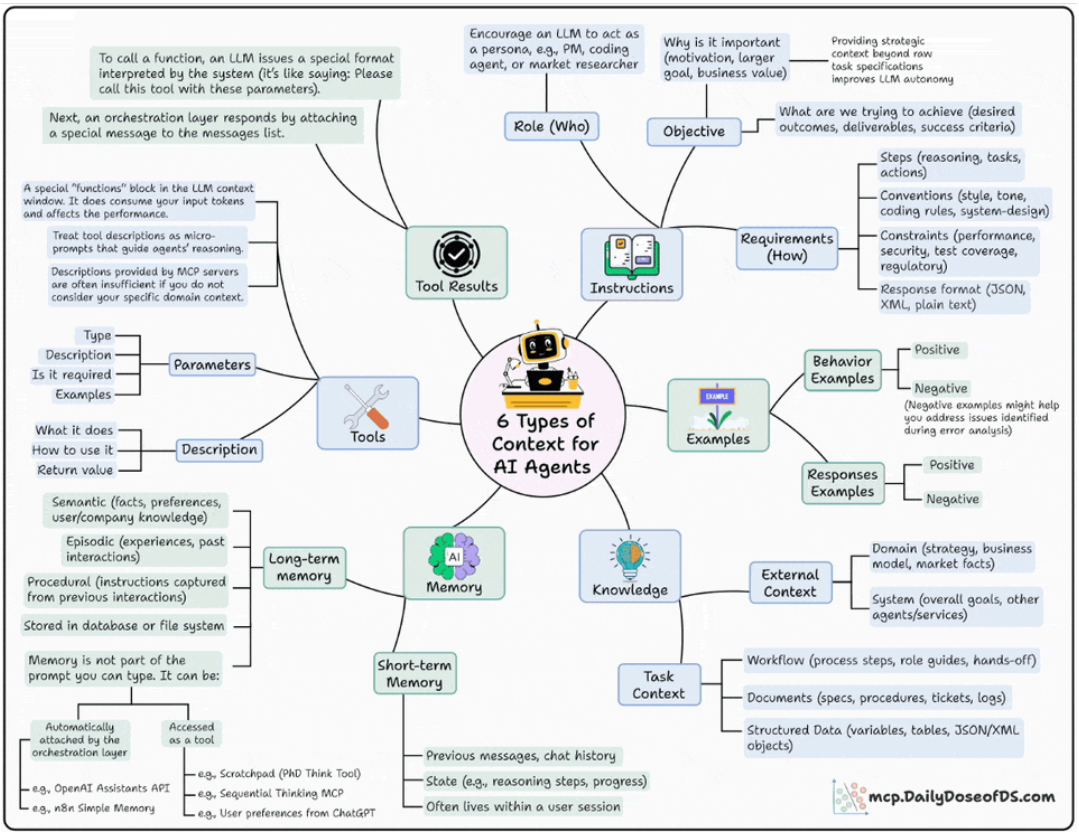
6 Types of Contexts for AI Agents

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.

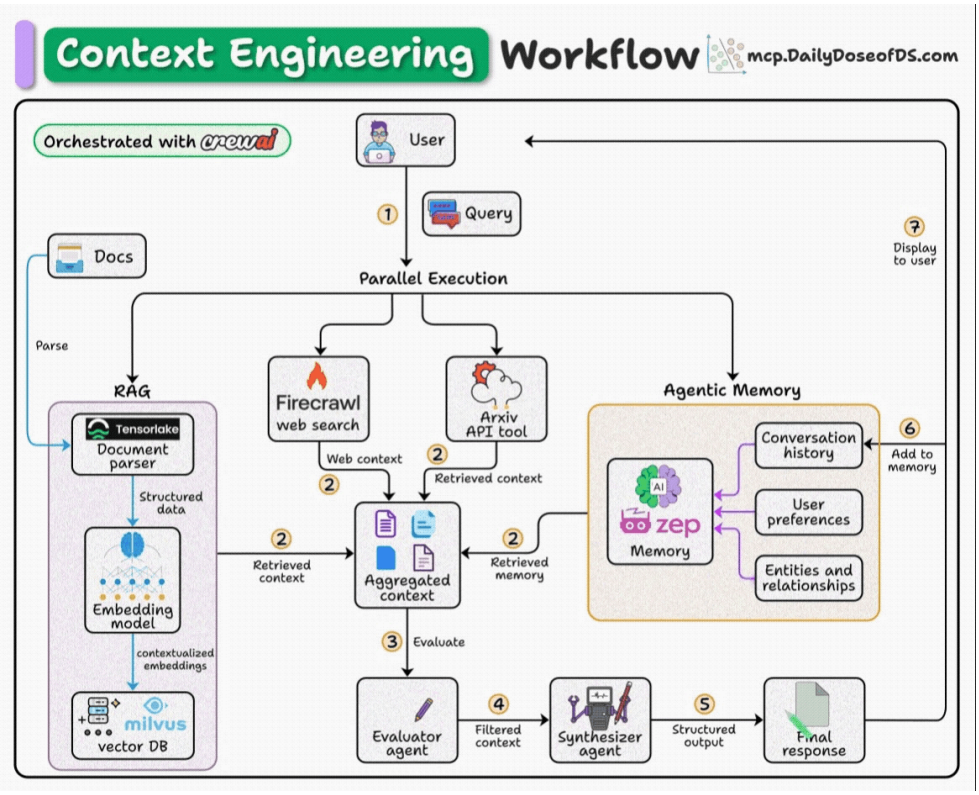
These are the exact six layers that help you build fully context-aware Agents.

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:

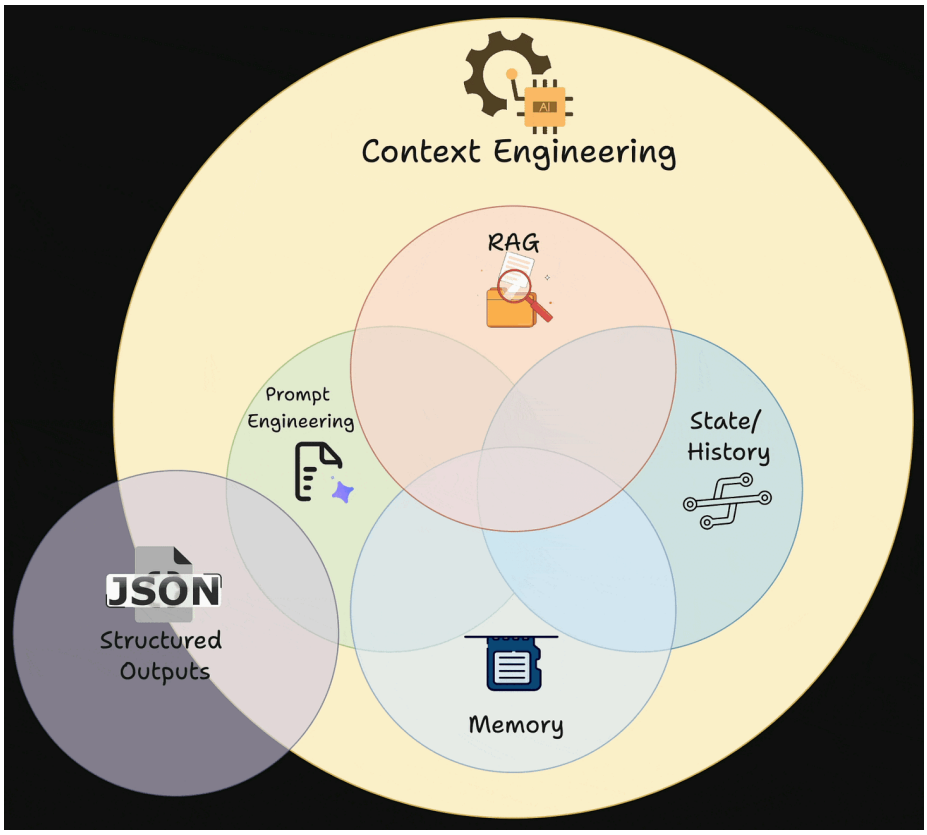


- User submits query.
- Fetch context from docs, web, arxiv API, and memory.
- Pass the aggregated context to an agent for filtering.
- Pass the filtered context to another agent to generate a response.
- Save the final response to memory.

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: