

A Guide to Using AI for Legal Research

From the Artificial Neuron to Large Language Models

Daria Godorozha

July 22, 2026

Technical descriptions and all legal and regulatory material in this guide are accurate as at 22 July 2026. This is a fast moving area, and both the technology and the law will have moved since; the details here should be verified against current sources rather than relied on as advice.

1 Foundations: How Machines Learn

1.1 1943: The McCulloch-Pitts Artificial Neuron

The story begins in 1943 with a simple mathematical model of a neuron. It takes binary inputs, sums them, and fires (outputs a 1) only if that sum crosses a fixed threshold. Crucially, it has no way to learn: its behaviour is set entirely by hand.

1.2 1957: The Perceptron (Rosenblatt)

The Perceptron, introduced by Rosenblatt in 1957, was a significant step forward. It accepts continuous, real valued inputs and learns a weight for each one, although its output is still binary, firing or staying silent depending on whether a threshold is crossed. What sets it apart from the McCulloch-Pitts neuron is that it can adjust its own behaviour through supervised learning. This is where the learning rule comes in: whenever the Perceptron gets a training example wrong, the rule nudges its weights in the right direction.

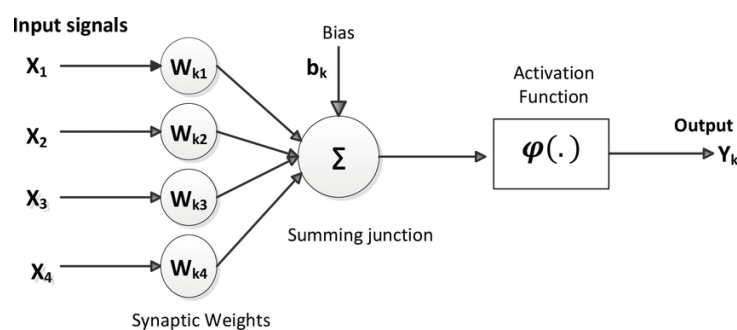


Image from Guesmi, Fathallah & Menif (2018), *Modulation Format Recognition Using Artificial Neural Networks for the Next Generation Optical Networks*, in *Advanced Applications for Artificial Neural Networks*, IntechOpen. Licensed under CC BY 4.0.

1.3 Three Ways a Model Can Learn

The Perceptron learns from labelled examples, but that is only one of three broad styles of learning worth knowing.

Supervised learning trains on labelled data, pairs of inputs and correct outputs, so the model learns to map one to the other. Classifying an email as spam or not spam, or predicting a house

price, are supervised tasks, and so is the digit classifier we build later in this guide.

Unsupervised learning works with unlabelled data and looks for structure the data already contains. Typical tasks include clustering similar items together, reducing many features down to a few that capture most of the variation, and spotting anomalies that do not fit the usual pattern.

Reinforcement learning (RL) is different again. Here an *agent* interacts with an *environment*, choosing an *action* in each *state* and receiving a *reward* as feedback, and it learns by trial and error to maximise its cumulative reward. The strategy it follows is called its *policy*. This vocabulary matters later, because reinforcement learning is one of the ways a modern language model is refined after its initial training.

1.4 Weights and Biases

Before going further, it is worth being clear about what an *input* is. An input is simply one piece of information you give the model to work with, translated into a number, usually from a dataset.

Think of the model as making a decision by weighing up evidence. Each input is a piece of evidence, and a *weight* says how much that piece should count, exactly as you might trust one friend’s restaurant recommendation more than another’s. A large positive weight means “this input strongly pushes the answer one way”, a weight near zero means “ignore this input”, and a negative weight means “this input pushes the other way”. The *bias* is a separate dial that shifts how easily the whole decision tips over, like a person who needs more convincing than most before they will say yes. These weights and biases are what the model *learns*; everything else about how it behaves follows from the particular values they settle on.

Nobody sets these numbers by hand, because there are far too many of them. Instead the model finds them by trial and improvement, a process called *gradient descent*. Picture standing on a foggy hillside wanting to reach the lowest point in the valley. You cannot see the whole landscape, but you can feel which way the ground slopes beneath your feet, so you take a small step downhill, then feel again, and repeat. Here the “height” is a measure of how wrong the model’s answers are, and each step nudges the weights and biases in the direction that makes the answers a little less wrong. Repeat this millions of times and the model gradually walks its way to values that work well.

Two more ideas explain why these models are so capable. First, because there are so many weights, it is convenient to organise them into grids of numbers called *matrices* (with the biases kept as simple lists called *vectors*), and the core operation each layer performs, combining every input with every weight, is just the arithmetic of multiplying these grids together. Second, after each layer multiplies its inputs by its weights, it passes the result through an *activation function* before handing it on. This function is a fixed rule that reshapes each number, most often by flattening negative values to zero and letting positive values through. That small step turns out to be essential. Multiplying by weights is a *linear* operation, meaning it can only scale and add its inputs, and chaining many linear operations together always simplifies to a single linear one, so a network built only from them, however deep, could do no more than a single layer: it could fit a straight line but never a curve. The activation function breaks that linearity. By bending the signal at every layer, it lets each layer transform the output of the one before it in a genuinely new way, and it is this repeated bending, stacked over many layers, that allows the network to represent curved, intricate patterns rather than straight line relationships alone.

A *vector*, in mathematics and physics, is a quantity that cannot be captured by a single number (a scalar). It has to be expressed by both a magnitude and a direction, and it is usually visualised as a point in n -dimensional space.

A *matrix*, in the context of LLMs, is a structured two dimensional array of scalars that serves as a compact way to represent data, transformations or relationships. An embedding matrix for a vocabulary is a good example: each row holds one word’s vector representation in a high dimensional space. When a matrix operates on a vector, by multiplying it as a linear transformation, it can stretch, shrink, rotate, reflect or shear that vector.

1.5 Feature Engineering

Classical machine learning depends heavily on *feature engineering*, the craft of turning raw data into informative input variables that help a model learn. Good features improve accuracy, generalise better to unseen data and can make training faster and more stable. One of the quiet revolutions of deep learning is that networks learn useful features for themselves rather than relying on a human to hand design them, and the embeddings we meet shortly are exactly this: features for language that the model discovers on its own.

1.6 1986: Backpropagation and Deep Learning

A major leap came in 1986, when Rumelhart, Hinton and Williams popularised the backpropagation algorithm for training multilayer networks. The technique was not invented outright that year, since it builds on earlier work in reverse mode differentiation and in training neural networks, but the 1986 paper is what established it as the practical way to make a network “deep”. Backpropagation is what makes training such a network possible.

To see why it is needed, recall that gradient descent improves the network by nudging each weight in the direction that reduces the error, which means it needs to know, for every single weight, how much that weight is to blame for the current error. In a deep network this is hard, because a weight buried in an early layer has no direct effect on the output at all. Its influence is entirely indirect: it changes the numbers passed to the next layer, which changes the numbers passed to the layer after that, and so on until, many steps later, the final answer shifts. Working out how a small change to that early weight ripples all the way through to the error is exactly the kind of question the *chain rule* from calculus answers. The chain rule says that when a change travels through a sequence of steps, its total effect is the product of the effects at each individual step, so the influence of a deep weight on the error is found by multiplying together the local sensitivities along the path from that weight to the output.

The remaining question is why backpropagation works *backwards*, from the output towards the input, rather than the other way round. The reason is efficiency, and it is the whole point of the algorithm. Every weight in a given layer shares the same downstream path to the output, so the sensitivities of all the later layers are common to them. If you started at the front and traced each weight forward on its own, you would recompute that shared downstream work over and over, once per weight, which for millions of weights is hopeless. Working backwards instead, the algorithm computes the error’s sensitivity at the output first, then passes it back one layer at a time, and at each layer it simply reuses the sensitivity already worked out for the layer behind it. In effect it distributes the blame for the error from the output back through the network, each layer taking the blame handed to it and splitting it among its own weights. This single backward sweep produces the gradient for every weight at once, which is what makes training a deep network practical rather than impossibly slow.

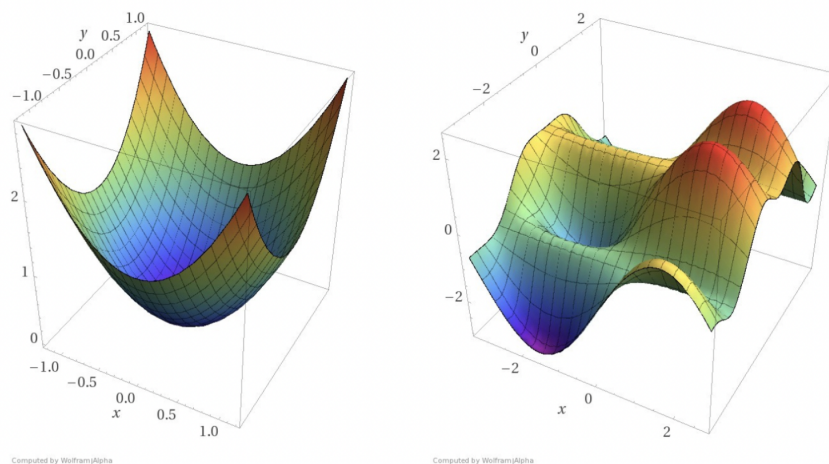
1.7 Efficiency and Optimisation

Representing these operations as matrix multiplications is what makes the models practical to run. Because a matrix multiplication applies the same operation across a whole batch of data at once, it can be computed in parallel, which is exactly what modern hardware like GPUs is

built for. This batch processing is what makes both training and inference efficient at scale. The same mathematical structure underpins training itself: backpropagation relies on matrix calculus to compute the gradients needed for gradient descent efficiently, layer by layer.

1.8 Gradient Descent

It helps to picture the error as a surface stretched out over the space of possible parameter values, with gradient descent as the process of walking downhill to find its lowest point to minimise the error. The update rule is simply $\theta_{t+1} = \theta_t - \alpha \nabla J(\theta_t)$, where θ are the parameters, J is the cost, ∇J its gradient and α the learning rate that sets the step size. In the friendliest case the surface is a single smooth bowl with one lowest point, and descent is guaranteed to reach it. Real networks give us something far bumpier, a landscape of ridges, valleys and many separate dips. There is a subtlety worth being honest about with students: because that surface extends far beyond anything we can directly measure, the minimum we actually reach is a local one rather than a guaranteed global minimum. What is surprising, and still not fully understood theoretically, is that these local minima tend to work remarkably well in practice. A learning rate that is too large overshoots the minimum, while one that is too small crawls, so tuning it is a key aspect.



1.9 The Training Loop

Training repeats a simple four step cycle. In the *forward pass* the model makes predictions from a batch of data. A *loss function* then measures how wrong those predictions are, using mean squared error for regression or cross-entropy for classification. The *backward pass* runs backpropagation to compute the gradient of the loss with respect to every parameter. Finally an *optimiser* updates the weights to reduce the loss, and the cycle repeats for many iterations. The optimiser is the component that decides how to turn gradients into weight updates. Stochastic gradient descent (SGD) takes a step against the gradient computed from a small random sample of data, which makes its path noisy but steadily downhill, while adaptive methods such as Adam and RMSProp adjust the step size for each parameter and tend to converge faster in practice.

The word *stochastic* here is worth pausing on, because it points to a source of unpredictability that matters beyond training. It refers to the deliberate use of randomness: the optimiser draws a different random sample of data at each step, the weights begin from a random initialisation, and the training data is shuffled into a random order. None of this is a flaw, since the randomness is part of what lets training escape poor solutions and generalise. But one consequence is that training is not a deterministic recipe. Run the same procedure on the same data twice and you can end up with two different models, with different weights and subtly different behaviour,

because each run followed a different noisy path down the loss surface.

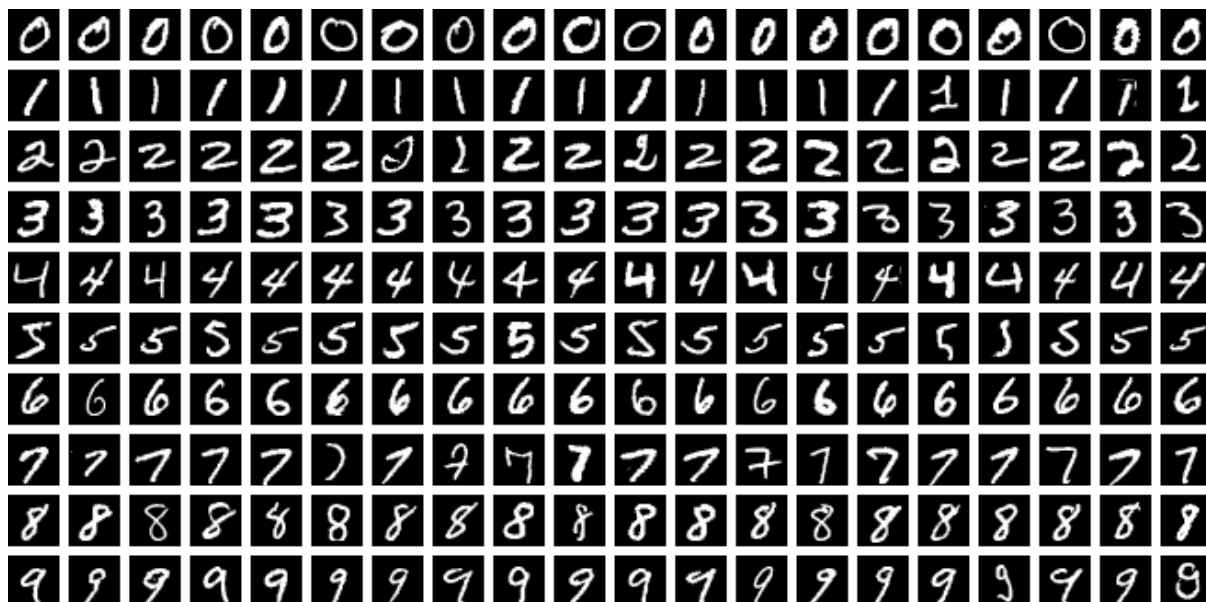
For a lawyer this has practical consequences. The model you rely on is not the unique, inevitable product of its training data; it is one draw from a range of models the same process could have produced, and you can neither see which draw you have nor reconstruct it. It matters more that when a provider retrains or updates a model, even on similar data, the result can behave differently from the version you tested last month, so a prompt that reliably produced a correct citation or a sound analysis may quietly start producing something weaker, with no announcement and no obvious signal. The risk is a false sense of stability. Treating a hosted model as a fixed, dependable instrument causes issue, because both the model itself and, as we will see when we turn to inference, the text it generates carry irreducible randomness. The professional response is to verify rather than assume and to keep a record of all prompts for verifiability.

1.10 The Softmax Function

Once a network has done its computation it usually produces a set of raw scores, called logits, one for each possible class. These logits are not normalised: they can be any real number, positive or negative, and on their own they do not represent probabilities. The softmax function is what bridges that gap, and it appears throughout modern AI for exactly this reason, including inside the attention mechanism we will meet later. It takes the raw logits and converts them into a set of probabilities that are all positive and sum to one, turning the model’s uninterpretable scores into something we can read directly as “how likely is each class”. For a vector of logits $z = (z_1, \dots, z_K)$, the softmax of the i th entry is

$$\text{softmax}(z)_i = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}}.$$

1.11 A Worked Example: Softmax on a Digit Classifier



A classic teaching dataset here is MNIST, a large collection of small grayscale images of hand-written digits from 0 to 9, each written many times over by different hands. Suppose we start with one such image of size 28 by 28 pixels. We flatten it into a single 784 dimensional vector and feed that into a neural network. The network’s final layer outputs ten logits, one for each

possible digit, for example

$$[1.2, 0.9, -0.5, 2.0, -1.1, 1.8, 0.3, -0.7, 0.6, -0.2].$$

The question is how to turn these raw scores into probabilities for each digit. Applying the softmax function does exactly that, giving approximately

$$[0.13, 0.10, 0.02, 0.30, 0.01, 0.25, 0.05, 0.02, 0.07, 0.03].$$

Each of these values is the likelihood that the input image is a particular digit. Here the digit “3” carries the highest probability, so the network predicts that the image is a “3”. This is supervised classification in miniature.

2 Representing Language as Numbers

In that last example the input was already numeric, since an image is simply a grid of pixel values. Language is not, so before a network can process text at all we need a way to turn words into vectors.

2.1 Representation of Data

In large language models, words, sentences, images and speech are all represented as vectors, often called embeddings, that live in a high dimensional space. These vectors are then stored and manipulated as the rows or columns of matrices, which, as we have just seen, is what lets us process them efficiently at scale.

2.2 Tokenization and Subword Embeddings

The first step in every language model is *tokenization*: breaking raw text into smaller pieces called tokens, each mapped to an integer ID using a fixed vocabulary. The choice of unit matters. Characters are too short and carry little meaning on their own, whole words are too many and cannot cope with words the model has never seen, so most models settle on subwords as the practical compromise. A phrase such as “unbelievable transformers” might split into “un”, “believable”, “transform” and “ers”, where a rare word breaks into familiar fragments and even the leading space can be part of a token. As a rough guide, one token is about three quarters of a word in English, so 100 tokens is roughly 75 words. Every model ships with its own tokenizer, and vocabularies typically run from around 30,000 to 200,000 tokens.

Once text is tokenized, an *embedding function* maps each token to a dense vector. In advanced GPTs like LLaMA 3, it is these subword tokens rather than full words that get embedded. The point of splitting words is worth restating: it lets the model handle unseen words through their sub-parts, keeps the vocabulary small and efficient, supports many languages at once, and can even help with spelling errors, which show up as unusual fragments. Each token is mapped to a dense vector in $\mathbb{R}^d$, where d is the embedding dimension (for instance, 8,192 in the 70 billion parameter version, 70B). The resulting embedding matrix for LLaMA 3 (70B) therefore has size

$$E \in \mathbb{R}^{128,000 \times 8192},$$

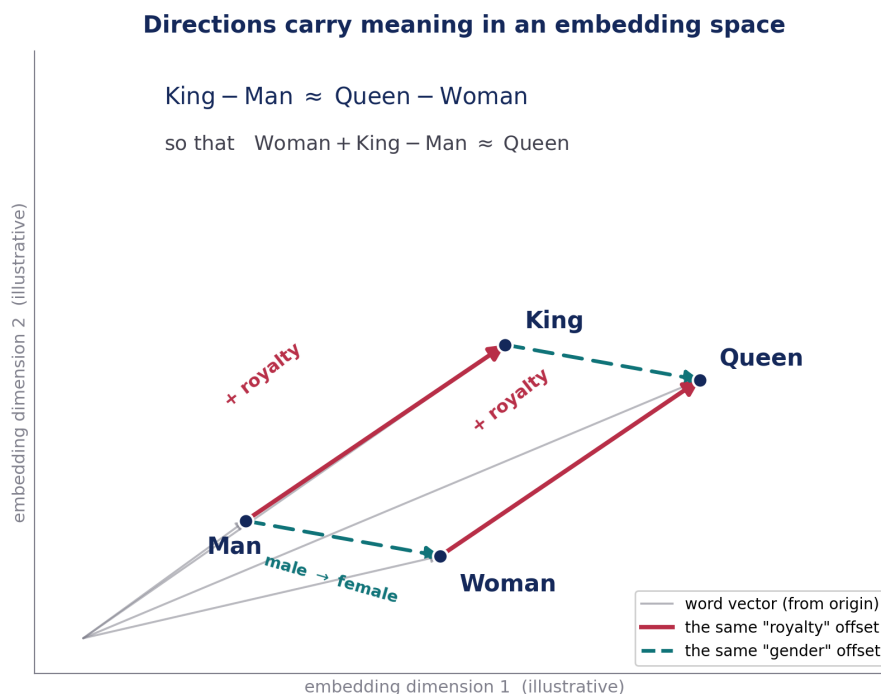
mapping each token to a vector of length 8,192.¹

¹Model dimensions, vocabulary sizes and parameter counts are quoted from the LLaMA 3 model card as published; they are specific to that release and differ across models and versions.

2.3 From Token IDs to Meaningful Vectors

It is worth being precise about what that vector does and does not capture. When tokens are first converted into discrete IDs (for instance, “bank” might become 1234), those IDs are arbitrary numbers with no semantic meaning of their own. The embedding matrix is best understood as a large learned lookup table with one row per token in the vocabulary: the token ID is used as a row index, and the corresponding row is that token’s embedding vector. This is a selection, not a computation. The rows start out random and are updated during training by backpropagation, which is why, in smaller models, the embedding matrix is often the single largest block of parameters.

A striking property of a well trained embedding space is that direction itself carries meaning. Take the vector for *King*, subtract the vector for *Man* and add the vector for *Woman*, and we land very close to the vector for *Queen*, written $\text{Woman} + \text{King} - \text{Man} \approx \text{Queen}$. Equivalently, the offset from *Man* to *King* is almost the same as the offset from *Woman* to *Queen*, so the two difference vectors point in nearly the same direction. This structure emerges purely from how the words are used, never from anyone labelling gender or royalty by hand.



A single fixed vector per token is powerful, but on its own it still cannot capture two things a model really needs. The first is the context of a token. A word like “bank” means different things in different sentences: in “river bank” it refers to the land beside water, while in “investment bank” it refers to a financial institution. Raw tokenization would assign the same token ID, and so the same starting vector, to “bank” in both cases, completely ignoring the distinction. Resolving this is precisely the job of the attention mechanism we will meet later.

The second is the position of a token. The model also needs to know where each token sits in the sentence, because order changes meaning. “Cats chase mice” says something quite different from “Mice chase cats”, even though both contain exactly the same tokens.

2.4 Adding a Sense of Position

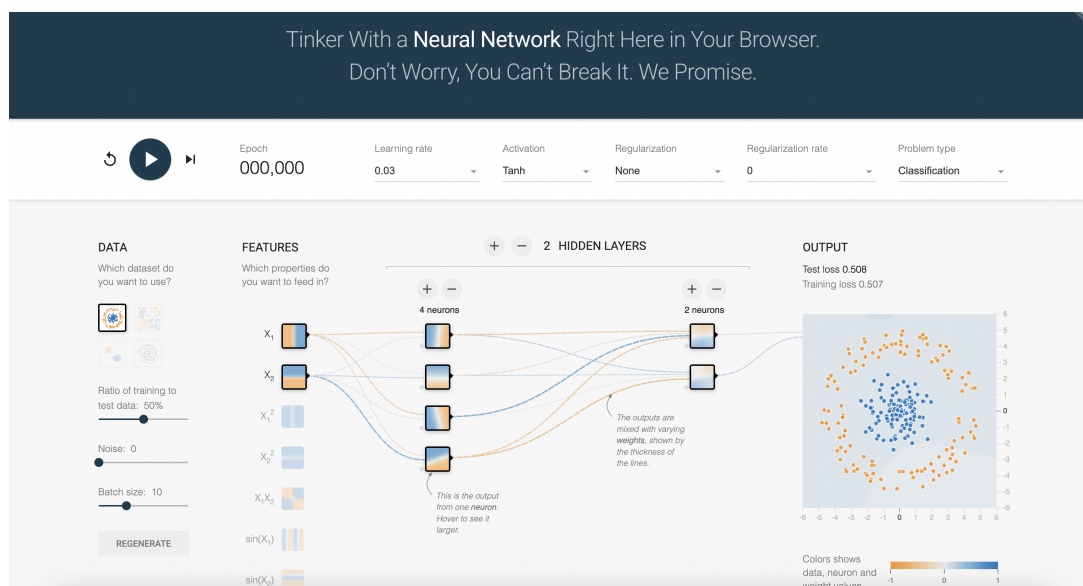
Positional encoding is what supplies that missing information about where each token occurs in the sequence. In the original Transformer, a position dependent vector was added to each token’s

representation, so that the same word appearing in two different places entered the network as two slightly different vectors. Many modern models instead use rotary positional embeddings, which do not adjust the token representations directly but rotate the query and key vectors used by attention, with the effect that attention scores become sensitive to the relative positions of tokens.

It is worth being clear about what this does and does not do. Positional encoding does not make neighbouring words more similar in meaning, and tokens that sit near one another do not simply become closer together in the vector space. What it provides is access to order, which is what allows the model to distinguish “Cats chase mice” from “Mice chase cats” when the two sentences contain identical tokens.

2.5 Building Intuition Interactively

A useful way to see these ideas in motion is the TensorFlow Playground at <http://playground.tensorflow.org>. It runs a small neural network directly in the browser and lets you change the learning rate, the activation function, the amount of regularisation and the number of hidden layers, then watch in real time as the network learns to separate a tangled two dimensional dataset. It is a low stakes way to develop a feel for what the weights and layers described above are actually doing.



3 Architectures That Changed the Field

With data represented as vectors and the basic mechanics of a network in place, we can turn to the architectures that put them to work.

3.1 Encoder-Decoder Networks with RNNs

Sequence data such as language posed a challenge, and the dominant early answer was the encoder-decoder built from recurrent neural networks (RNNs). What distinguishes RNNs is that they carry memory from previous time steps in the form of a hidden state. Recall, too, that an embedding layer is what maps the input tokens to the dense vector representations these networks operate on.

The **encoder**’s job is to encode a sequence of vectors, which in this example are the embedding vectors of the input tokens. As it processes the sequence, it produces a hidden state h for every

position, and the full set of these encoder states is written $H = \{h_1, \dots, h_s\}$.

The **decoder** takes a context vector c , which can simply be the encoder’s final hidden state, and uses it to initialise its own first hidden state d_0 . From there the recurrent decoder acts as a conditional language model, modelling two things at once: fluency, which comes naturally from its intrinsic language model nature, and adequacy, which it achieves by conditioning on the source representation.

The flow is easiest to follow with a translation example. The encoder reads the embedded English tokens “ $\langle s \rangle$ I love dogs $\langle /s \rangle$ ”, producing hidden states h_1 through h_5 . The final state becomes the context vector, $c = h_5$, which initialises the decoder. The decoder then generates the German translation one token at a time, “Ich mag Hunde”, where each step passes through an output layer and softmax and feeds its own prediction back in as the input to the next step. The weakness of this design is that the entire input has to be squeezed into a single fixed length context vector, which becomes a bottleneck for long sequences.

3.2 The Transformer (2017)

The Transformer, introduced in 2017 in the paper “Attention Is All You Need”, was designed to remove that bottleneck, and in doing so it revolutionised natural language processing. Instead of compressing everything into one vector, its central innovation, the attention mechanism, lets the model look directly at every input position. Attention enriches token vectors by folding in contextual information, allowing the model to identify relationships between the current word and every other word in the sequence and so work out which words matter most for predicting the next one. This is also what finally resolves the context problem raised earlier, since the vector for “bank” can now be shaped by the words around it.

The mechanism leans on a geometric fact we have already used: direction carries meaning, and the dot product between two vectors is large when they are aligned in the vector space, indicating similarity, and falls to zero when they are orthogonal. Attention does not, however, compare the raw embeddings directly. It is built from three learned matrices, the Query, Key and Value, which project each token’s representation into three separate spaces, and it works by comparing each token’s query projection against every other token’s key projection. By passing these comparison scores through a softmax function, the model turns them into a probability distribution that highlights which tokens are most relevant to the query token, then uses those weights to blend the values. Running several such attention operations in parallel, known as multi-head attention, lets the model attend to different kinds of relationship at once. The result is a set of updated representations that encode how relevant every word is to every other word in the sequence. Each token’s original information is not lost along the way, but that is largely down to the residual connections that carry the input of each sub-layer around it and add it back to the output, rather than to attention on its own.

A well known example makes this concrete. In the sentence “The animal didn’t cross the street because it was too tired”, attention correctly links the word “it” most strongly to “animal”. Change a single word, so that the sentence reads “...because it was too wide”, and the same mechanism now links “it” most strongly to “street”. The model has resolved the pronoun by attending to the word that makes sense of the whole sentence.

This attention based approach largely displaced RNNs and LSTMs in mainstream natural language processing, giving models a much better grasp of context in language. It transformed foundational AI research and led to state of the art results in both language understanding and generation, and it paved the way for the billion parameter models we see today.

3.3 Mixture of Experts (MoE)

As transformers grew, a question arose: how do we keep adding capacity without every token paying the full cost of a bigger model? Mixture of Experts is one answer. An MoE layer holds many parallel sub-networks called experts, but a small router, or gating network, sends each token to only a few of them, typically the top two. The outputs of the selected experts are then combined by a weighted sum, so that in spirit the layer computes $\sum_i \text{gate}(x)_i \cdot \text{expert}_i(x)$ over just the chosen experts. Because only a small subset fires for any given token, the model can hold far more total parameters, and specialise them, while keeping the compute and memory cost of each step low.

3.4 Beyond Attention: State Space Models

Attention is powerful but expensive, since comparing every token with every other costs on the order of N^2 for a sequence of length N , which becomes painful beyond tens of thousands of tokens. State space models (SSMs) offer an alternative for very long sequences. They treat a sequence as a dynamical system, maintaining a hidden state h_t that evolves over time according to $h_t = Ah_{t-1} + Bx_t$ and produces outputs $y_t = Ch_t + Dx_t$, with A , B , C and D learned. This structure scales linearly with sequence length rather than quadratically, which makes SSMs attractive for tasks with very long context, from long document understanding to time series and genomics.

4 A Shift in How Models Are Built

Having seen the key architectures, we can step back to how the practice of building models changed once the Transformer arrived.

4.1 The Old Way

Training AI models used to demand a great deal upfront. It relied on large, carefully curated datasets, along with significant compute resources and time. It also called for real expertise across a range of tasks: data preprocessing, algorithm selection, hyperparameter tuning, feature engineering and more. Every new problem tended to mean a new model built more or less from scratch.

4.2 The New Way

The arrival of foundation models has transformed this. Rather than one model doing everything, it is the family of foundation models that collectively spans the modalities: GPT and LLaMA are language models, DALL-E generates images from text, and some recent models are natively multimodal, handling text, images and audio within a single system. What they share is the pattern that matters here. Each is pre-trained once, at great expense, on a very large dataset, and can then be reused across many tasks. Users no longer need to train a model from scratch; instead they simply fine tune an existing one for their own task or use it directly for inference, whether through a hosted API or by running an open weight model on their own hardware.

5 Training a Modern Large Language Model

Most systems of this kind are built on a *decoder only* transformer. The original Transformer paired an encoder that reads an input with a decoder that produces an output, which suits translation, whereas a decoder only model keeps just the generating half and produces text by repeatedly predicting the next token. LLaMA 3 is a documented example of this design, and it

is the architecture behind most current chat oriented language models, though it is worth being careful with attribution: “ChatGPT” is a product whose underlying models change over time, and Claude’s full architecture has not been publicly specified. It is safer to say that this family of designs dominates current systems than to attribute a specific architecture to a particular product.

5.1 Pre-training at Scale: LLaMA 3

Training starts by converting a large multilingual text corpus into discrete tokens and then pre-training the model on that data to perform next token prediction. During this stage the model learns the structure of language and absorbs a great deal of knowledge about the world simply from the text it reads. The scale is considerable: the largest model has 405 billion parameters and is pre-trained on around 15.6 trillion tokens with a context window of 8,000 tokens, later extended through continued pre-training to 128,000 tokens.²

5.2 How the Weights Are Calibrated

The weights begin as random numbers, so at first the model’s output is nonsense. Calibrating them means running the training loop from earlier over and over on real text: the model predicts the next word, the loss measures how far off it was, backpropagation works out how much each weight contributed to that error, and the optimiser nudges every weight a little in the direction that would have made the answer better. One pass teaches the model almost nothing, but repeated across trillions of words and billions of tiny adjustments, the weights gradually settle into values that encode the grammar, facts and patterns of language. Nobody chooses these numbers or can read meaning off any single one; they are simply whatever values make the model good at prediction. This is the mechanism. What differs between the stages below is not how the weights are adjusted, which is always this loop, but what the model is being trained to predict or prefer.

Pre-training gives the model a broad understanding of linguistic patterns, languages, grammar and semantics. It has four main components: curating and filtering a large scale training corpus; developing the model architecture and the scaling laws that determine model size; developing techniques for efficient pre-training at scale; and developing a pre-training recipe that ties these together.

Post training then adapts the pre-trained model to align better with human preferences and with the way people actually use language. A typical alignment pipeline collects prompts, generates several candidate responses per prompt, and scores them with a reward model. Rejection sampling keeps the stronger responses to form supervised fine tuning data, which produces a supervised fine tuned model. A further preference optimisation step, such as DPO, compares pairs of responses and pushes the model towards the preferred one, yielding the final aligned model.

5.3 Reinforcement Learning: Group Relative Policy Optimisation

Reinforcement learning, whose vocabulary of agents, actions, rewards and policies we saw earlier, offers another route to improving a model after pre-training, and the DeepSeek work provides a clear example in Group Relative Policy Optimisation (GRPO).

Recall that a *policy* maps each possible state to a probability distribution over the actions available in that state. Here the state is a given prompt and the action is the next token, or more loosely the next possible response. For a language model the policy is therefore its own

²Figures as reported in the LLaMA 3 paper. Training scale and context windows change with every model generation, so these should be read as an illustration of the order of magnitude rather than as current maxima.

internal probability distribution over all the token sequences it might generate given a prompt, which means that adjusting the model’s parameters is really a way of reshaping that distribution so that desirable, high reward responses become more likely.

GRPO is designed to cut the cost of doing this. Most reinforcement learning methods train a second network, a critic, to estimate how good a response is, and that critic is typically the same size as the policy model itself, which roughly doubles the expense. GRPO foregoes the critic altogether and estimates its baseline from group scores instead. Specifically, for each question q it samples a group of outputs $\{o_1, o_2, \dots, o_G\}$ from the old policy $\pi_{\theta_{old}}$, and then optimises the policy model π_θ by maximising the objective

$$\mathcal{J}_{GRPO}(\theta) = \mathbb{E}[q \sim P(Q), \{o_i\}_{i=1}^G \sim \pi_{\theta_{old}}(O | q)] \\ \frac{1}{G} \sum_{i=1}^G \left(\min \left(\frac{\pi_\theta(o_i | q)}{\pi_{\theta_{old}}(o_i | q)} A_i, \text{clip} \left(\frac{\pi_\theta(o_i | q)}{\pi_{\theta_{old}}(o_i | q)}, 1 - \epsilon, 1 + \epsilon \right) A_i \right) - \beta \mathbb{D}_{KL}(\pi_\theta \| \pi_{ref}) \right),$$

where the divergence term is

$$\mathbb{D}_{KL}(\pi_\theta \| \pi_{ref}) = \frac{\pi_{ref}(o_i | q)}{\pi_\theta(o_i | q)} - \log \frac{\pi_{ref}(o_i | q)}{\pi_\theta(o_i | q)} - 1,$$

and where ϵ and β are hyperparameters and A_i is the advantage, computed using the group of rewards $\{r_1, r_2, \dots, r_G\}$ corresponding to the outputs within each group:

$$A_i = \frac{r_i - \text{mean}(\{r_1, r_2, \dots, r_G\})}{\text{std}(\{r_1, r_2, \dots, r_G\})}.$$

Each piece has an intuitive reading. The ratio $\pi_\theta(o_i | q)/\pi_{\theta_{old}}(o_i | q)$ measures how much more the current policy prefers a response than the old policy did, and multiplying it by the advantage is what encourages the model to assign better probabilities to better responses, meaning those with higher A_i . The clip is a stability device: by capping that ratio to the range $1 - \epsilon$ to $1 + \epsilon$ it ensures updates are not too large and prevents the model from over correcting in a single step, limiting how far the policy can deviate at once. The KL term is a divergence penalty, weighted by β , which ensures the model does not stray too far from a reference distribution π_{ref} , usually the model it started from. Finally the advantage A_i is what makes the method *group relative*: the model computes an advantage for the reward r_i of each response o_i by subtracting the mean reward of its group and dividing by the standard deviation, so that $A_i > 0$ means the response is better than the average of its group and $A_i < 0$ means it is worse. The group itself supplies the baseline that a critic network would otherwise have had to estimate.³

5.4 Regularisation

A concern that runs through all of this training is generalisation. In machine learning, regularisation is used to prevent overfitting, a problem where the model fits too closely to the training data but cannot generalise to new, unseen data. Additionally, regularisation can help introduce sparsity by reducing the number of active features, which in turn requires less computation and fewer resources. In practice this is often implemented as a norm penalty added to the loss, which is why regularisation is sometimes described as the use of norms in practice.

Overfitting is not just a technical inconvenience, and for legal research it carries a particular danger. A model that has fitted too closely to its training data has, in effect, memorised surface patterns rather than learned the underlying principles, so it performs impressively on

³Formulation as set out in the DeepSeek-R1 technical report, which adopts GRPO from Shao and colleagues (2024). See https://github.com/deepseek-ai/DeepSeek-R1/blob/main/DeepSeek_R1.pdf.

material that resembles what it has seen and then fails on anything genuinely novel. In law this is exactly the wrong failure mode. The questions that matter are often the ones at the edges: an unusual set of facts, a niche jurisdiction, a point on which the authorities are thin or conflicting. An overfitted model tends to answer these by pattern matching to the closest thing in its training data, producing something that reads fluently and looks like sound analysis but is really a familiar template stretched over an unfamiliar problem. Worse, it fails with confidence rather than hesitation, because a memorised pattern comes out just as smoothly as genuine understanding, so there is no drop in tone to warn the reader that the model has left solid ground.

A distinct risk, easily confused with overfitting but different in kind, is temporal. A model can state a rule that has since been amended or overruled, with the same assurance as the current position. This is not a failure to generalise from training data to unseen data, which is what overfitting means; it is a knowledge cutoff or data freshness problem, since the model learned the law as it stood when its training data was collected and has no way of knowing what has changed since. The two call for different remedies. Overfitting is addressed during training, through regularisation and honest evaluation on held out data, whereas staleness is addressed at the point of use, by grounding the model in current sources through retrieval and by checking that any authority it offers is still good law. What they share is the practical implication that runs through the rest of this guide. Fluency is not reliability, an answer that fits the pattern is not the same as an answer that fits the law, and novel or high stakes questions are precisely where independent verification against primary sources matters most.

5.5 From Text to Reasoning Models

These training ideas turn a text predictor into something closer to a reasoning system. Two levers do much of the work. **Chain of thought** prompts the model to lay out intermediate steps rather than jumping straight to an answer, which improves accuracy on problems that need several stages of inference. **Long context** gives the model a much larger window to read at once, so it can hold whole documents, transcripts or codebases in view while it reasons. Together they let a model work through complex, multi-step questions of exactly the kind that arise in legal analysis.

The same logic that makes chain of thought work inside a single answer applies, even more strongly, across a whole workflow. Asking a model to research an issue, analyse it and write the summary in one shot forces every stage to happen invisibly in a single pass, where you cannot see or check the intermediate steps and where one early mistake silently contaminates everything after it, which is the cascading error problem again. Decomposing the task into a sequence of smaller prompts, each with a narrow job and a checkable output, is far more reliable. Each step can be verified before the next builds on it, and the points where the model might hallucinate are isolated and contained. Because the randomness discussed earlier means any single generation is only one draw, running a step several times and comparing the results is also informative, though it should be read carefully: divergence between runs is a useful warning that the model is on shaky ground, while agreement is weaker evidence than it appears, since repeated runs share the same model, the same training data and the same retrieved material and can therefore reproduce the same systematic error. Consistency is a prompt to look closer, never a substitute for checking the source. The cost is more effort and more tokens, but for anything that will be relied on, a chain of small verified steps beats one confident paragraph.

A concrete example is worth spelling out, because it shows how to keep such a workflow anchored to facts rather than to the model's fluency. Suppose the task is a summary of a discrete legal issue. Rather than ask for the summary directly, split it into stages. *First, find the literature:* have the model, ideally with retrieval or search tools, identify the relevant statutes, cases and

commentary, treating this only as a list of candidates and not as settled sources. *Second, build a database of that literature*, a simple table with one row per source recording its citation, date, jurisdiction and a link or file reference. *Third, verify before going further*: independently confirm that each source actually exists, says what it is claimed to say and is still good law, since this is the step that catches fabricated citations and superseded authority before they can propagate. *Fourth, extract the specific points* you need from the verified sources, working from the documents themselves rather than the model’s memory of them, so that each extracted proposition is drawn from a passage you can point to. *Fifth, record provenance*, extending the table into a spreadsheet in which every extracted fact sits beside the exact source and passage it came from, so that the evidential basis of the eventual summary is visible and auditable at a glance. *Only then, sixth, write the summary*, drafting it from that provenance spreadsheet rather than from a free form request, so that every sentence traces back to a checked source.

This ordering inverts the naive approach, and deliberately so. The model is used where it is strong, at finding, reading, extracting and drafting, but never as the final authority on what the law is, because that authority rests on primary sources you have checked. The provenance spreadsheet is the load bearing artefact: it makes the summary reproducible and defensible, since a colleague, a client or a court can trace any statement back to its origin, and if the underlying model changes the facts do not, because they live in the table rather than in the model.

6 How a Model Generates Text

Once a model is trained, using it to produce text is a separate process called inference. It is worth understanding in its own right, because it explains both the behaviour you see and the cost you pay.

6.1 Inference Is a Forward Pass

Generating text is a forward pass through the network. Your prompt is tokenized into integer IDs, those tokens travel through the stack of transformer layers as attention and feed forward maths, and the model comes out the other side with a probability distribution over the next token. That token is chosen, appended to the input, and the whole process runs again for the next one. Because generation happens one token at a time in this way, the model is called autoregressive.

6.2 Prefill and Decode

Inference has two distinct phases with very different speeds. In the *prefill* phase the model reads the whole prompt at once, processing all input tokens in parallel in a single batch, which is fast. In the *decode* phase it then produces output one token per step, sequentially, and this is the part you see streaming onto the screen a word at a time. The shift from a fast parallel read to a slow sequential write is why a response often starts after a pause and then flows steadily.

6.3 The KV Cache

Every token generated is added back to the context and influences the next, which threatens to make each step redo all the attention work of the tokens before it. The *KV cache* avoids this by storing the key and value vectors already computed for earlier tokens, so the model does not recompute them. The saving can be substantial, though how substantial varies a great deal with the architecture, the sequence length, the batch size and the implementation. It comes at the cost of extra memory, which is one reason a larger context window demands more RAM or VRAM.

6.4 Sampling the Next Token

The model scores every token in its vocabulary, but a sampling strategy decides which one is actually emitted. *Temperature* scales how sharp or flat the distribution is: a low value makes the model focused and repetitive, a high value makes it more creative and more random. *Top-p* sampling keeps only the smallest set of tokens whose probabilities sum to p , trimming the unlikely tail. *Top-k* keeps only the k highest scoring tokens. *Greedy* decoding always takes the single most probable token, which is deterministic but can fall into repetition. For legal work, where reproducibility and precision matter, a low temperature is usually the safer default, though, as we will see shortly, even a temperature of zero does not make a hosted model perfectly reproducible.

6.5 When Generation Stops

A model does not inherently know when it is finished; it stops according to explicit criteria. The main ones are an end of sequence token that the model itself generates to signal completion, a maximum token limit set by the caller as a hard cap, custom stop sequences that halt generation when a particular string appears, and simple user interruption. Setting these sensibly is part of controlling both output length and cost.

6.6 Hardware and Quantisation

A single forward pass through a frontier model can require an enormous number of arithmetic operations. GPUs can perform the large parallel matrix operations used in inference far faster than conventional CPUs, which is why inference at scale depends on them, although actual response times vary widely with model size, quantisation, hardware, prompt length, batching and whether one is measuring a single token, the time to the first token or the complete answer. Memory is the other constraint, since a 70B model needs tens of gigabytes just to hold its weights. *Quantisation* eases this by storing weights at lower precision, for example compressing them from 16 bit to 4 bit numbers. The quality loss is usually small, and the payoff is large: with 4 bit quantisation a smaller model can run on a laptop rather than a data centre GPU. In practice, tools such as Ollama package this quantisation so that running an open weight model locally takes a single command, a point we return to in the practical toolkit.

6.7 Why Temperature Zero Is Not Determinism

It is tempting to think that setting the temperature to zero makes a model deterministic. Greedy decoding does remove the sampling randomness, since with no temperature and no truncation the model simply takes the single highest scoring token every time, and that step contains no random choice. Yet running the same prompt twice, even at temperature zero on the same machine, can still produce different text. The randomness has two separate sources. The algorithmic kind, controlled by temperature, top-p, top-k and the random seed, is intentional and does vanish at temperature zero. A second, systemic kind survives, and it comes from how the numbers are physically computed.

The root cause is that floating point addition is not associative: because every operation is rounded to a fixed number of digits, adding the same numbers in a different order can give a different result. In 32-bit arithmetic, for instance, $(10^{20} + 1) - 10^{20}$ evaluates to 0, because the 1 is lost to rounding, whereas $10^{20} + (1 - 10^{20})$ evaluates to 1. Producing a single token involves billions of such additions across many layers, and the order in which a GPU adds them is chosen for speed, not for reproducibility. Several things can change that order between two runs. The library or driver may pick a different routine for the same matrix multiplication, and the lower precision formats used at serving time round more coarsely. Most importantly in production,

your request is batched together with other users' requests to make efficient use of the hardware, and because the kernels are not batch invariant, the number of strangers sharing your batch changes the tiling and summation order, and so the last few digits of the logits. In mixture of experts models the effect is starker still: experts have a fixed capacity, so on a busy server your token can be bumped from its first choice expert to its second, a genuinely different function that returns different scores. Models split across many GPUs add a further twist, since the partial results must be combined over the network in whatever order happens to be fastest.

None of these perturbations is large; they typically sit below one part in a million. They matter only when two candidate tokens are almost tied, in which case a rounding difference far too small to see can flip which one wins. And once a single token flips, the damage does not stay local. Because each new token is generated conditioned on all the tokens before it, one changed token early in an answer sends the two runs down different paths, and the texts diverge in structure, length and content, much like the sensitive dependence on initial conditions seen in a chaotic system. In one striking demonstration, researchers at Thinking Machines Lab sent an identical prompt a thousand times, at temperature zero, to Qwen3-235B-A22B-Instruct-2507, and obtained eighty distinct completions; swapping in batch invariant kernels produced a thousand identical outputs.⁴ The one line summary is that temperature zero makes the decoding rule deterministic given fixed logits, but it does not make the logits, and therefore the output, identical from one run to the next.

6.8 The Economics of Tokens

Because both the prompt and the response are measured in tokens, tokens are the unit of cost when using a model through an API. A simple way to compare providers is tokens per dollar: if a model charges $\$p$ per thousand tokens, then one dollar buys $1/p$ blocks of a thousand tokens, which is $1000/p$ tokens in total. At one cent per thousand tokens, for instance, a dollar buys a hundred blocks, or 100,000 tokens. Input and output tokens are often priced differently, and cheaper models can offer far more tokens per dollar than frontier ones, which is why matching the model to the task matters. Costs can be reduced by being concise in prompts, summarising or chunking long documents, capping output length, and caching, since a cached prompt can be served far more cheaply than one recomputed from scratch. For a practice watching its budget, more tokens per dollar simply means more room to build and test.

7 Grounding, Retrieval and Agents

A trained model on its own is fluent but sealed off from current facts.

7.1 Retrieval Augmented Generation (RAG)

Retrieval augmented generation addresses a basic limitation of a standalone language model, namely that its knowledge is frozen at training time and it can invent details when asked about things it has not seen. RAG pairs the model with an external store of documents. The documents are broken into chunks, converted into embeddings and stored in a vector database. At query time the question is embedded too, the most relevant chunks are retrieved by similarity, and they are placed in the model's context so that the answer is grounded in specific source material rather than in the model's parameters alone. For legal work this is particularly valuable, because it lets

⁴Reported by Thinking Machines Lab, *Defeating Nondeterminism in LLM Inference* (2025). In their unoptimised prototype the deterministic kernels took roughly 42 seconds against 26 seconds for the standard ones, about 62 per cent greater latency, corresponding to roughly 38 per cent lower throughput. Their separate performance comparisons used a smaller model.

a model draw on a defined corpus of statutes, cases and internal documents and, in principle, point back to where each statement came from.

Simple RAG retrieves once and then generates. *Advanced* RAG turns the process into a loop: the system plans, retrieves, reasons over what it found, optionally calls tools, and refines its plan across several steps before answering, which improves accuracy on harder questions at the cost of more time and expense. Retrieval quality itself can be tuned, for example by combining keyword and vector search, re-ranking the retrieved passages with a stronger model, or preferring more recent sources.

What an API is

All of this assumes the documents can actually be fetched, and in practice they are fetched through an *API*. An API, or application programming interface, is simply a defined way for one program to ask another for something and get a structured answer back. Where a person visits a website and reads a page, a program sends a request to an API, a precise message such as “give me the full text of this case” or “return every judgment citing this statute”, and receives the answer as clean, machine readable data rather than as a web page meant for human eyes. The API is best thought of as a service counter with a published menu: it lists exactly what you may ask for and in what form, it usually requires a key that identifies you and tracks your usage, and it often limits how many requests you may make in a given period. For a retrieval system this matters because APIs are what let it pull current, authoritative documents on demand instead of relying on a static snapshot, and the model’s tools, described later, are in essence just API calls the model is allowed to make.

Connecting to legal databases

Connecting a retrieval system to legal sources therefore means connecting it to their APIs, and the sources fall into a few groups. A growing number of primary materials are available through official or open APIs: in the United Kingdom, legislation.gov.uk and the National Archives’ Find Case Law service expose statutes and judgments in structured form, and comparable government and court services exist elsewhere, alongside open projects that publish case law as data. These are the natural first building blocks, because primary law is where grounding matters most. The major commercial platforms, the Westlaw and LexisNexis family, hold the richest content, including editorial headnotes, annotations and citators, but access to their data through an API is governed by a licence rather than freely available, and increasingly they offer their own retrieval and AI features built on top of that content. A firm’s own material, its matter files, precedents and know how, is the third source, reached not through a public API but through connectors to its document management and internal systems. In each case the engineering pattern is the same as the RAG pipeline above: the connector fetches documents through the API, they are chunked and embedded into the vector store, and retrieval proceeds as before. The practical differences are authentication, cost and freshness, since API keys must be kept secret, metered calls cost money, and a retrieval system is only as current as the last time it fetched from the source.

Using data lawfully

The final consideration is the one most easily overlooked in the rush to build: that data is accessible does not mean it is free to use as you wish. Three limits bear on a legal RAG system in particular. The first is licensing and terms of use. Content from a subscription platform is licensed for defined purposes, and loading it wholesale into your own vector store, or using it to train or fine tune a model, may well exceed what the licence permits, so the terms have to be read before the data is ingested rather than after. Automated collection deserves special care:

scraping a database that an API or its terms did not invite you to collect from can breach those terms even where the underlying material is public. The second is copyright, discussed later in this guide, which attaches to editorial and compiled content such as headnotes and annotations even where the primary law beneath them is free to reproduce, so a licensed commentary is not yours to redistribute simply because you can retrieve it. The third is confidentiality and data protection. Client and personal data carry duties of privilege and, under regimes such as the UK GDPR, of lawful processing, so where such material is embedded, and in particular whether it is sent to a third party model or embedding service, is a question to settle deliberately, which is one of the reasons the local, self hosted stack described later is attractive for confidential corpora.

7.2 Structured Retrieval: Knowledge Graphs

A vector database is not the only place to retrieve from. A *knowledge graph* stores explicit relationships between entities, such as “Nolan directed Inception”, as a network of nodes and links governed by a schema, or ontology, that defines which entities and relationships are allowed. The two tools answer different questions. A vector database answers similarity questions (“what is similar to this passage?”) in a fuzzy, probabilistic way, ranking results by closeness. A knowledge graph answers structured questions (“which films did this director make with this actor?”) deterministically, returning the same answer every time it is asked of the same database. That consistency should not be mistaken for correctness, however, since the answer is only ever as accurate and as complete as the graph behind it, which depends in turn on the quality of the source data, the design of the ontology and the entity extraction that populated it. Building one means extracting entities and relationships from documents, defining a schema so that the relationships make sense, and storing the result in a graph database. For law, where citation networks, corporate structures and statutory cross-references are inherently relational, knowledge graphs are a natural fit alongside similarity search.

7.3 From Models to Agents: the ReAct Loop

A raw model gives one answer and stops. It cannot verify a fact, call a tool, or recover when it is wrong, which is not enough for real tasks. An *agent* closes this gap by wrapping the model in a loop. The most influential pattern, introduced by the ReAct work in 2022, interleaves reasoning and acting: the model has a *thought* about what to do next, takes an *action* by calling a tool, receives an *observation* from that tool, and then thinks again, repeating until the task is done or a step limit is reached. This matters because chain of thought reasoning alone is stuck inside the model’s head, whereas ReAct grounds each step in real results. Its properties are what make it the basis of most serious agent systems: the actions taken and the observations returned are inspectable, so there is a record of what the agent actually did, the observations come from real tools rather than the model’s imagination, and the number of steps flexes with the difficulty of the task. One caution belongs here. The written reasoning between those steps is itself generated text, and it need not be a complete or faithful account of how the model actually arrived at its choice, so the tool calls and their results are the auditable part, not the narration around them.

7.4 Tools and the Model Context Protocol

An agent is only as capable as its tools. If the model is the brain that reasons and decides, tools are the hands that let it act: web search for current facts, code execution for running and debugging code, and databases, APIs and file systems for everything else. Anything that can be called as a function can become a tool. The *Model Context Protocol* (MCP) is an open standard for connecting a model to such external tools and services, from email and code repositories to design tools, in a uniform way. A practical rule is to keep each tool focused and clearly described,

since vague descriptions lead the agent to reach for the wrong one.

7.5 Memory and Multi-Agent Systems

Two further ingredients turn a demo into something usable. *Memory* lets an agent persist information across sessions, so it does not reset and force the user to re-explain context every time; without it, an assistant is effectively blind to what it did last time. *Multi-agent systems* split a large task across several specialised agents coordinated by an orchestrator, which helps when a single agent would overflow its context window or when different sub-tasks call for different tools. These systems only work well when handoffs and shared state are handled deliberately rather than left as an afterthought.

8 Building With These Ideas: A Practical Toolkit

The concepts in this guide are not confined to research papers; they are realised in a mature open ecosystem of tools. Two are worth knowing as starting points, one for running a model and one for building an application around it. The wider landscape is broad, taking in LlamaIndex for retrieval, vLLM for high throughput serving and the Hugging Face libraries for models and datasets, but the two below are enough to make the ideas concrete.

8.1 Running Models Locally with Ollama

Ollama is an open source tool that downloads and runs open weight models, such as Llama, Mistral, Gemma and Qwen, directly on your own computer. A single command like `ollama run llama3` pulls the model and starts a chat, and the same install exposes a local HTTP API that applications can call. Under the surface it relies on the quantised formats described earlier, which is exactly what allows a capable model to run on a laptop rather than a data centre GPU.

For legal work the appeal is less about novelty than about control. Running the model on your own machine means the text you send it need not be transmitted to a third party, which matters for client confidentiality and privilege. That protection is not automatic, however, and it holds only if the whole pipeline is local: the embedding model, the vector store, any tools the system calls, and any telemetry or logging must all stay in house, since a single remote component can send data out of the building even when the model itself does not. There is no per-token charge, it works offline, and, since you own the entire stack, it is the most practical route towards the reproducibility discussed later, though pinning the model and its configuration is not on its own enough to guarantee identical outputs, which also requires making the decoding settings and the numerical execution stack deterministic. The trade off is capability, since open weight models that fit on a laptop are generally smaller and less able than the frontier hosted models, so the sensible course is to match the tool to the task and reserve local models for work where privacy or cost outweighs raw power.

8.2 Adapting a Model Without Retraining It

Fine tuning a full downloadable model updates every parameter, which is expensive. Parameter efficient methods change far less while still specialising the model for a task. *LoRA* (low rank adaptation) freezes the pre-trained weights and inserts small trainable matrices alongside them, so that a weight matrix W is effectively replaced by $W + \frac{\alpha}{r}BA$, where B and A are low rank and r is a small rank. Often fewer than one per cent of the parameters are trained, which means lower cost, faster training, smaller saved checkpoints and the ability to keep many task specific adapters for one base model. Two lighter techniques go further still by freezing the model entirely and training only a small set of continuous vectors, though they differ in where those vectors act. *Prompt tuning* optimises continuous embeddings that are inserted at the model's input, in

effect learning a soft prompt in place of a written one. *Prefix tuning* instead introduces trainable continuous prefixes into the attention computation itself, commonly as layer specific key and value states, so it steers the model from inside each layer rather than only at the input. These techniques are what make “the new way”, adapting a foundation model rather than building one, practical for individual teams.

For a legal team the first practical question is not how to fine tune but whether to. These methods sit on a ladder, and the cheaper rungs solve most problems. A clear system prompt and a few worked examples in the prompt itself cost nothing to change and are enough for many tasks. Retrieval augmented generation, covered earlier, is the right tool whenever the goal is to ground answers in your own statutes, cases or documents. The crucial point that is widely misunderstood is what fine tuning actually teaches. It shapes *behaviour*, such as a house drafting style, a consistent citation format, or a reliable way of classifying or extracting information, but it is a poor and risky way to inject *facts*. Trying to teach a model your case law by fine tuning tends to produce fluent invention rather than dependable recall, because the model learns the shape of the material without a faithful copy of it. Facts belong in retrieval; fine tuning is for form and task, so most researchers should reserve it for narrow, repetitive, well defined jobs where the output has a consistent structure.

Where LoRA does earn its place, the workflow is within reach of a small team. You assemble a dataset of input and output pairs that demonstrate the task, typically a few hundred to a few thousand examples, such as a clause paired with its risk classification, a judgment paired with a structured summary, or an instruction paired with a passage written in your firm’s style. You choose an open weight base model of the kind that runs under a tool like Ollama, then use an established library to train the adapter: the Hugging Face PEFT library is the standard building block, and higher level tools such as Axolotl or Unsloth wrap it into a simpler recipe. A quantised variant known as QLoRA lets you fine tune a surprisingly large model on a single GPU, whether one you own or one rented by the hour, which keeps the cost modest. Crucially, you hold back a portion of your examples as a test set the model never sees during training, and you judge success on that held-out set rather than on the training data, which is the direct practical guard against the overfitting described earlier. The finished adapter is a small file that you load on top of the base model at inference time, and you can keep several, one per task, against the same base.

Two cautions matter more in law than in most fields. The first is confidentiality: your training examples may contain privileged or client material, so it matters where the training actually runs, and privileged data should not be uploaded to a third party fine tuning service without checking what that service may retain or reuse. Running the process in your own or a trusted environment, as the local stack described above allows, keeps the data in house. The second is provenance: record which base model, which dataset and which adapter version produced a given behaviour, because a fine tuned system is only reproducible and defensible if you can say exactly what went into it, and because a change to the underlying base model can shift the adapter’s behaviour and warrants re-testing.

8.3 Orchestrating Applications with LangChain

Where Ollama runs a model, LangChain helps you build something around one. It is a framework, available in Python and JavaScript, whose building blocks map almost one to one onto the concepts in this guide: a uniform interface to many providers so that a hosted model and a local one behind Ollama can be swapped freely, prompt templates, connectors to vector databases for retrieval augmented generation, tool integrations, memory, and ready made agent loops. Its companion library LangGraph expresses an agent as a graph with explicit state, which fits the cyclic think act observe loop and the multi agent orchestration described earlier, and it makes

each step observable and testable rather than hidden inside a single call.

The value of such a framework is that it turns the ideas of this guide, retrieval, the ReAct loop, tools and memory, into a few composable components instead of bespoke glue code, and its evaluation and tracing tools support the repeatable testing that both benchmarking and reproducibility demand. A word of caution belongs here too: a framework is a convenience, not a guarantee of correctness.

8.4 A Minimal Local Stack for Legal Research

The tools combine naturally into a small, self contained pipeline that illustrates the whole arc of this guide. A firm's documents are split into chunks and turned into embeddings by a model running under Ollama, then stored in a local vector database. A question is embedded the same way, the closest chunks are retrieved, and LangChain assembles them with the question into a prompt that a local model answers, citing the passages it used. Provided every component is genuinely local, including the embedding model and the store, nothing leaves the firm's own hardware, the running costs are fixed rather than per query, and the setup is far easier to reproduce because every part of it is pinned and under one roof. It is a modest system, but it exercises tokenization, embeddings, retrieval, generation and grounding together, and it is a realistic first step for a practice that wants to keep a confidential corpus in house.

9 Using AI Responsibly in Legal Practice

9.1 Interpretability

Retrieval and tools help us trust an answer by grounding it in sources, but they do not tell us what is happening inside the model. A live research question is whether we can look in directly and understand how a model represents concepts. Anthropic's work on mapping the internal features of a large language model is a good entry point and is described at <https://www.anthropic.com/research/mapping-mind-language-model>.

That work identifies internal features that correspond to a remarkable range of concepts, from cities and named people to chemical elements, scientific fields and even elements of programming syntax. These features are multimodal and multilingual: the same feature can respond to an image of a thing as well as to its name or description across many languages.

The features are not merely observed but can be turned up or down. When researchers amplify a feature tied to a particular landmark, the model begins steering almost every reply towards that landmark, even mentioning it in answers where it has no relevance, to the point of describing its own physical form in those terms. This kind of feature steering is a vivid demonstration that specific, human interpretable directions inside the model causally shape its output.

9.2 Sycophancy

The same steering can expose less charming behaviour. Turning up a feature associated with flattering praise makes the model gushingly sycophantic, showering a mundane prompt with extravagant compliments rather than useful feedback. This is not only a laboratory curiosity. Research reported in early 2026 found that leading chatbots remain markedly sycophantic by default, and framed this as a widespread behaviour with real downstream effects on users rather than a narrow or purely stylistic concern. For a lawyer relying on such a tool, a model that tells you what you want to hear is a genuine risk, and it leads directly into the questions of reliability and responsibility.

9.3 Bias

Bias in these systems is subtle. One counter intuitive point is that models tend to agree with each other, and with a confident user, more than you might expect, which is a natural consequence of how they are trained on overlapping data and optimised to produce probable, agreeable continuations. That apparent consensus can give a false sense of reliability, since several tools echoing the same answer are not independent witnesses to the truth. It is also the same tendency, seen from another angle, that produced the sycophancy described above.

9.4 Hallucination

No single failure mode matters more to a legal researcher than hallucination, the model's habit of stating confident falsehoods, so it is worth drawing together here what the rest of this guide has established about where it comes from and what to do about it.

It is tempting to blame hallucination on the randomness in how a model samples its words, but that gets the causation the wrong way round. The tendency comes principally from what the model learned, not from how it samples. As we saw when the weights were calibrated, a language model is trained to produce fluent, plausible continuations of text, and nothing in that objective gives it a sense of truth or a reliable signal for what it does not know. So when it lacks a fact, it does not fall silent. It generates the most plausible looking answer, which can be an invention delivered in exactly the same assured tone as a correct answer. This happens even at temperature zero, with the sampling randomness switched off entirely, which shows that random sampling is not necessary for hallucination. That is not the same as showing sampling to be irrelevant: decoding settings such as temperature can still affect how often fabrications appear and how they read.

There is a deeper reason the problem is stubborn. The model has no separate store of facts to consult; its knowledge is smeared across billions of weights as statistical patterns, so recalling a real case and fabricating a plausible sounding one are, mechanically, the same act of generation. A citation that does not exist can be produced with the same fluency as one that does, because the model is not looking anything up. This also compounds the temporal problem noted earlier: the model reproduces the patterns present when it was trained, so it can state a rule that has since been amended or overruled with complete confidence.

Hallucination is hazardous in every field, but law sharpens the danger for two reasons. The first is the confidence. A fabrication arrives with no change in tone, no hedging and no visible seam, so there is nothing in the text itself to warn the reader that the model has left solid ground. Fluency reads as authority, and a plausible invention is precisely the kind of error that slips through a busy review. The second is the stakes and the form. Legal work runs on specific, checkable claims, that a case says a particular thing, that a statute is in force, that an authority supports a proposition, and these are exactly the claims a model is most likely to get subtly wrong while sounding right. The most notorious examples are fabricated case citations, complete with realistic parties, courts and dates, that turn out to refer to nothing at all, and lawyers in several jurisdictions have already faced sanction for filing them.

The randomness discussed earlier is not the root cause of hallucination, but it does make it harder to manage. Because generation can follow a different path on each run, a model may invent a citation on one attempt and not the next, or invent a different one each time, so the same underlying weakness surfaces inconsistently. Catching a fabrication once is therefore no guarantee the next run is clean, which is why a single spot check is not a safeguard.

Nothing removes the tendency entirely, so the goal is to contain it. The measures that work are the ones this guide has returned to throughout. Ground the model in real sources rather than its memory: retrieval, which places actual documents in front of the model, is the single

most effective structural defence, though it reduces rather than eliminates the problem, since a model can still misread or overstate a source it has been given. Verify every specific, checkable claim against the primary material, treating any case, citation or quotation as unconfirmed until you have seen it yourself, because this is the step that catches fabricated authority before it propagates. Decompose important work into small, checkable stages rather than asking for a finished answer in one pass, so that an invention is caught early instead of being built upon. And record provenance, so that every claim in a deliverable can be traced back to a source you have checked. Underlying all of these is a single professional habit: treat fluency as a property of the writing, never as evidence of truth. The model is an able assistant for finding, reading and drafting, but the responsibility for what is actually true remains, as it always has, with the lawyer.

9.5 Liability

As these tools move into professional settings, the question of who is responsible when they go wrong becomes pressing. In healthcare, for example, the Medical Protection Society warned in June 2026 that doctors and the NHS could face medical negligence claims over mistakes made by AI tools used to diagnose patients and suggest treatment, and called for the law to be reformed so that clinicians are not left carrying liability for errors introduced by the technology. That is a professional body's warning about potential exposure and a call for reform rather than a judicially settled rule, but the parallel for legal practice is direct: using an AI tool does not transfer professional responsibility away from the practitioner, and the governing rules are still catching up.

9.6 Cascading Errors

A particular danger in legal work is that a small mistake made early can multiply as it moves through the process. Picture a piece of work moving through three hands. A junior associate doing basic research identifies the wrong case or rule, so the error originates. A senior associate then builds analysis and drafting on top of that flawed foundation, accepting the mistaken premise, so the error amplifies. By the time a partner reviews the near final brief, the error has multiplied and now infects the key arguments and conclusions, demanding more corrections, weakening the position and raising both cost and risk. The lesson is simple to state and hard to live by: accuracy at the start and rigour throughout produce better outcomes, whereas a single unverified citation near the beginning can quietly compromise everything built on it.

9.7 Reproducibility

The way these models generate text, described earlier, has a direct consequence for legal work: the same query can return different answers on different runs, and when using a hosted service you cannot fully prevent it. This is more than an annoyance. Reproducibility underpins evaluation, auditing and any fair comparison between tools, and it matters whenever a result may later need to be defended or re-checked. An answer that cannot be reproduced is hard to verify for a client or a court, and, as with cascading errors, a single token that flips early can change which authority a model cites and rewrite the analysis that follows.

The practical response is not to trust a single run. Where an answer matters, run the same query several times and look at how far the responses agree, treating wide variation as a warning. Agreement, though, is weaker evidence than it looks: repeated runs draw on the same model, the same training data and the same retrieved passages, so they can reproduce the same systematic error with perfect consistency, and verification against an external source therefore remains necessary whatever the runs agree on. For evaluation and audit, report results over repeated samples with some measure of spread rather than quoting a single figure. Where exact

reproducibility is genuinely required, it can be engineered, but usually only on a self hosted setup with pinned software versions, fixed batch shapes and deterministic kernels and decoding, and at some cost in speed; even then, complete reproducibility can depend on the hardware, the software versions and whether deterministic operations have been enabled, as the PyTorch documentation itself warns. On shared commercial endpoints most of these controls are out of the user’s hands, which is why their seed settings are documented as best effort rather than as guarantees. Determinism, in short, is an achievable engineering target, not a free property that comes from setting the temperature to zero.

9.8 Benchmarking LLMs

All of this makes careful evaluation essential rather than optional. Benchmarking a language model means testing it against defined tasks and datasets so that claims about its accuracy, reasoning or reliability can be checked rather than assumed. Agents in particular should be tested like software, with repeatable evaluations rather than one off impressions. For legal applications the benchmarks that matter are the ones that reflect real practice: correctly identifying governing authority, reasoning faithfully from sources, resisting the pull towards agreeable but wrong answers, and flagging uncertainty instead of hiding it. Choosing and interpreting these benchmarks well is itself part of using AI responsibly.

A lawyer does not have to run these tests personally, because several public sources already do. It helps to separate two kinds. *Academic benchmarks* probe the underlying model: LegalBench, a collaboratively built suite of 162 tasks grouped into six categories of legal reasoning, measures legal reasoning; LexGLUE covers legal language understanding; CUAD tests clause extraction in contract review; and newer suites such as LegalAgentBench, which concentrates on the Chinese legal domain, evaluate tool using agents rather than bare models. These are mostly hosted on arXiv and GitHub and are useful for understanding a model’s raw capabilities and limits. *Practitioner facing evaluations* test the actual products a firm might buy. The most cited is the Stanford RegLab and HAI study *Hallucination Free? Assessing the Reliability of Leading AI Legal Research Tools* (Magesh and colleagues), which found that even retrieval based tools from established vendors hallucinated on a meaningful share of queries, contradicting marketing claims of hallucination free citations, and which named sycophancy as one of its four error types. The independent evaluator Vals AI runs a public legal leaderboard and issues periodic Legal AI Reports comparing named commercial tools on tasks such as document extraction, summarisation, redlining and legal research. University law library guides, for instance those maintained by several US law schools, curate and summarise these studies and are a good neutral starting point.

Read all of them with the cautions this guide has stressed. Every such result is a snapshot of particular model and tool versions at a particular date, so a figure from last year may describe a system that no longer exists, and the numbers move even as the underlying lesson does not. No single leaderboard is authoritative, rankings differ by task and rubric, and a vendor’s own published benchmark is a marketing document until an independent party reproduces it. Most importantly, a strong benchmark score is evidence about a tool in general, not a substitute for verifying the specific answer in front of you, which remains your professional responsibility regardless of how well the tool scores.

9.9 Plagiarism and Copyright

A generative model produces text by recombining patterns from an enormous training corpus, and this raises two distinct intellectual property questions that a practitioner should keep separate.

The first is whether the *output* infringes someone else’s copyright. Because a model has, in a

loose sense, read protected works, it can on occasion reproduce passages that are recognisably close to its training data, a phenomenon sometimes called regurgitation or memorisation. This is not hypothetical: a central allegation in the ongoing New York Times litigation against OpenAI is that the model can emit passages closely mirroring original articles, and the Getty Images case against Stability AI points to generated images carrying garbled Getty watermarks. Fluent output that looks original may therefore contain material that is not, and, unlike a human author, the model gives no signal that it is quoting. The professional risk is plagiarism by accident: incorporating model text into a brief, article or client deliverable without realising it tracks a protected source. The mitigations are the ones this guide has stressed throughout, applied to a new end. Treat model output as a draft to be checked rather than as original work, run substantial passages through the plagiarism and source checking tools a firm already uses, and, for anything that leans on a specific source, work from the primary material and cite it directly rather than relying on the model's paraphrase.

The second question is whether *you own* what the model helps you produce, and here the law distinguishes sharply between AI generated and AI assisted work. In the United States the position is now reasonably settled at the level of principle: copyright requires a human author, a rule the D.C. Circuit affirmed in *Thaler v. Perlmutter* and which the Supreme Court declined to review on 2 March 2026. Purely AI generated material is not protectable, while AI assisted work can be, but only to the extent a human exercised meaningful creative control over the expressive elements, and the Copyright Office has indicated that prompts alone will generally not supply sufficient human control for authorship. The United Kingdom is an unusual outlier, since section 9(3) of the Copyright, Designs and Patents Act 1988 provides for computer generated works with no human author, attributing them to the person who made the arrangements necessary for their creation, though how this interacts with the post Brexit originality standard remains untested. The practical consequence is that a document generated substantially by a model may enjoy weaker protection than one a lawyer wrote, which matters for client deliverables and for anything a firm intends to treat as its own valuable work product.

Underlying both questions is a fast moving and unsettled area of law. The training data cases have so far produced divergent outcomes even within a single jurisdiction, some courts finding AI training to be fair use on the facts before them and others rejecting it where the output competes directly with the source, and the first appellate rulings are only now emerging. In the United Kingdom, the government consulted on reforming the text and data mining exception and, in a statutory report and economic impact assessment published in March 2026 under the Data (Use and Access) Act 2025, declined to adopt any of the options consulted on, including the broad exception with a rightsholder opt-out it had previously favoured. The practical effect is that the existing position stands for now, so commercial use of protected works to train a model in the United Kingdom still generally requires a licence, with further evidence gathering and possible reform signalled rather than legislated. None of this is settled law, and the specifics will have moved by the time this is read, so the details here should be treated as a snapshot as at 22 July 2026 to be verified against current authority rather than as advice to rely on.