Under the Hood

Sapiens and Large Language Models Compared

Bryant Cruse & Rosaire Leon

How sapiens work

Components of a sapiens

The sapiens platform consists of an application development environment and a runtime architecture.

The development environment, named MELD, is the toolkit a developer uses to construct a sapiens. It supports modeling work, logic development in the MICA programming language, automated testing, and debugging. MELD can generate multiple runtime sapiens instances, and each instance can operate a different version of a model. As a sapiens processes inputs at runtime, it creates graph structures. MELD displays those structures in detail, both as itemized lists and as visual representations, and it permits the developer to inspect and debug each step of the MICA program that created them.

A running sapiens has four components: the engine; the world model; the reasoning routines that process the model; and whatever interfaces the application requires. The engine loads the model and the reasoning routines at runtime and stores them in system memory. A user or a connected system then submits structured information to the sapiens, whether natural language messages or any stream of ordered data. A sapiens can accept several such streams at once; they needn’t be of the same kind.

Every sapiens application shares a baseline model. The first part of that baseline is the Cognitive Core, a meta-knowledge schema that classifies every model component, from the most elemental to the most complex, according to how components may combine to form more complex ideas. The Cognitive Core also constrains those combinations, so that whatever a sapiens assembles remains consistent with the model it already holds. The functionality of the Cognitive Core can be compared to scientific chemistry’s periodic table of the elements, which specifies how atoms can combine to form molecules and substances.

The second part of the baseline is the commonsense world model, which holds the foundational concepts from which a modeler assembles application-specific or expert knowledge. The commonsense world model also supports natural language comprehension, for applications that require a natural language interface. It currently holds roughly three thousand concepts. The platform remains in development.

How a sapiens acquires its model

A sapiens’ world model originates in the mind of a modeler, a human expert in the domain of knowledge that the model will reconstitute in synthetic form. Starting from the domain’s most elementary concepts, MELD guides the modeler through a twelve-

step process. That process applies to each concept the epistemological and ontological properties that the Cognitive Core defines, and it ensures that the concept remains compatible with the system's processing logic.

One such property is a pointer towards a more abstract concept, called a genus parent. The concept under construction inherits its generic properties from that more abstract concept, while the pointer itself fixes the new concept's address in the overall model. Other properties distinguish the new concept from its genus parent and from the other children of that same parent. The modeler may then connect defined concepts to one another, so that the functional model of the entire domain accumulates.

New Sapience calls this process *synthetic intelligence engineering*. It is not teaching, as when people exchange ideas through language; the modeler declares each essential property explicitly rather than leaving the sapiens to infer it. Nor is it programming: what the modeler produces is an information structure, not a processing routine.

What happens when you use it

A sapiens transcribes each incoming stream, whether statements in a natural human language, telemetry from instrumentation, or data from a database, into an intermediate structure called an input graph. The reasoning routines then transform each input graph, according to the logic and organizing principles of the Cognitive Core, into extensions of the model the sapiens already holds. This process is analogous to human cognitive learning, in contrast to the "Machine Learning" that calibrates an Artificial Neural Network's coefficients.

The Cognitive Core is the "DNA" of synthetic knowledge: it is a compact specification protocol that enables a well-bounded and stable body of code to transform minimally sufficient quantities of information into functional models of the world.

Natural-language input passes through this process as follows.

1. A user submits a message in natural language.
2. The sapiens transcribes the message into a directed graph structure, the input graph, which the MICA reasoning routines can process. The input graph stores all the grammatical and syntactical information from the original message.
3. Where a word in the message is defined in the sapiens language model and carries a world model reference, the sapiens forms a link from the input graph directly into the existing model structure. These links connect the symbols, which are words, to the non-linguistic model structures, which are concepts.
4. The sapiens then builds a second graph, the concept graph, by analyzing the connection properties of those model references. It reads the grammar and syntax of the input graph as instructions for how to assemble the new idea from the model referents.

5. The sapiens analyzes the connection properties of the referents to determine whether the assembly those instructions call for is valid. A concept graph derived from (for example) “My horse read a book” is rejected at this step.
6. The sapiens transforms and refines the concept graph as required. If the input graph captures what someone *said*, the concept graph is an attempt to capture what they *meant*. What someone says may be plausibly understood in terms of the model and still not be exactly what they meant to convey. Read most literally, “I took my medicine” means the speaker picked it up; the sapiens possesses logic that automatically transforms the concept graph from the action *take* to the action *ingest*.
7. The sapiens creates model referents for unknown words in the input graph, inferring them from the connection properties of the known references in the concept graph. From “Librarians read books,” the sapiens infers that *librarians* are *people*, because a connection property of *reading* is that it can only be predicated of *people*. It adds the word *librarian* to the system with the model reference “a set of people,” and it applies information from every subsequent reference to *librarians* to that new model element.
8. The sapiens analyzes what the knowledge in the concept graph implies for the larger world model, and applies those implications. If *changing location* is a defined effect of the action *traveling*, and the concept graph specifies that someone *traveled* to *Chicago*, the sapiens updates that person’s location to *Chicago*. In many cases, the defined effect of an action is another action with its own effect, which the sapiens also asserts. The downstream effects of modeled actions thus form chains of causality, which the sapiens retains and can subsequently use to solve problems and answer questions.

How a sapiens uses knowledge of cause and effect

Suppose the input is: *I leave ice cream out on the counter. What happens?*

1. The first statement is comprehended as described above. The household context is recognized, and the location of the substance *ice cream* is updated to the *kitchen counter*.
2. The question is processed as described above.
3. *Happens* maps to a query for an effect of the action *leaving*, with the object *ice cream*. The sapiens calls its reasoning routines for causal analysis and passes them these model elements.
4. The initial effect of *leaving* is already known: the ice cream is *on the kitchen counter*. No other direct actions on the ice cream have been described as input, so

the routine looks for cause-and-effect relationships that attend the change in location.

5. Examining this instance of *ice cream*, the sapiens determines that it belongs to the set *frozen desserts*, solid substances kept in *freezers* to prevent *thawing* or *melting*, which suggests possible outcomes when such desserts are removed and *left out on the counter*.
6. Ice cream is known to be composed largely of *milk* and *cream*, substances which are *liquid* at *room temperature*.
7. *Kitchens* are known to be at *room temperature*.
8. *Melting*, the process of *changing state* from *solid* to *liquid*, is recognized as the answer to the input question, which the sapiens articulates in English as “It melts.”

The logic is general and reusable. It applies equally to anything that undergoes a state change due to a change in ambient conditions, with the particulars as specified in the model. The sapiens solved this problem at the level of common everyday knowledge because it recognized the query belonged to the household context. Had the context been more technical, and the knowledge available, the sapiens could have solved the problem just as well in terms of (for example) thermodynamics and molecular chemistry, which are themselves models that describe what happens to the ice crystals within the ice cream as they absorb heat.

What you can inspect

The functionality of a sapiens application derives from the logic contained in its MICA routines and from the structure of the model on which that logic operates.

MICA is a fully deterministic procedural computer language. Given the same input or inputs, each execution step can have one and only one result. MELD permits developers to examine every execution step of a runtime operation, with full visibility into each input and each result.

Human modelers create the baseline model structure explicitly, and it is fully readable and editable within MELD. Some of the model’s persistent storage formats are themselves human-readable and human-editable. MELD also gives developers full visibility into the model structure that the runtime program creates, in both itemized lists and visual representations.

How Large Language Models (LLMs) work

Large Language Models ("LLMs"), also called generative AI, are the technology that underlies the current generation of popular AI chatbots (ChatGPT, Claude, Gemini, Grok, etc.).

An LLM is created by using algorithms to convert an enormous quantity of written language, authored by human beings, into a model composed of statistics about how fragments of that writing appeared in order relative to one another.

Components of a Large Language Model

At the most basic level, an LLM consists of two permanent components and several more components that vary depending on whether we are considering the LLM in training or deployment. The two permanent components are an Artificial Neural Network ("network") and a separate program called a tokenizer.

The tokenizer breaks digital text into smaller pieces. Digital text is already a sequence of numbers, one or more for each character. A token is a further number that stands for a short word, a fragment of a longer word, a punctuation mark, or a special symbol. The tokenizer has an index of around a hundred thousand tokens. It uses that index to convert input text into a sequence of tokens, and later to convert an output sequence back into output text. That is all the tokenizer does.

A token has no connection to whatever the original word meant. Meaning—the symbolic connection that ties signs to their real references—is not found in the text itself; meaning *was* intended by the human beings who authored the text and *is* intended each time by those who read and interpret it. Simply put, text is not inherently meaningful; language and knowledge are two different things, and language alone does not constitute or even convey knowledge. To draw knowledge from language requires more than a faculty for measuring patterns within that language; it requires the ability to interpret signs *as signs*, as representations of things intended, and the awareness that such representations are distinct from the things themselves. (The word "apple" is not an apple.)

Tokens derived from text are no more inherently meaningful than the text itself was: drawing knowledge from them also requires a faculty of interpretation. Human beings (and, in a deterministic way, sapiens) possess this faculty, but the network is an arithmetic function; it has no faculty for interpreting the tokens it handles. As input, it receives a sequence of tokens, signs isolated from any reference. As output, it returns a list of numbers, one for each token in the index. Each number measures the probability that its token is the one that follows the input sequence.

The arithmetic function includes many billions of coefficients. Each coefficient multiplies one of the numbers that the function computes from the input sequence. The products accumulate to yield the function's output list, the probability distribution that the LLM uses to process user input. When a developer reports a model's size in billions

of *parameters* or *weights*, that number represents how many coefficients the arithmetic function uses. The coefficients begin as random numbers and are adjusted through the training process; they remain constant thereafter for the end-user. The input sequence varies with every use.

The network needs other programs to operate it. These are the components that vary between different stages of training and final deployment. During pre-training and supervised fine-tuning, a program performs a statistical operation called *curve-fitting*, which adjusts the network's coefficients. During reinforcement learning, a second program performs a statistical operation called *sampling*, which generates candidate outputs; a second network called the *reward model* scores those outputs; and a reinforcement program adjusts the first network's coefficients based on those scores. In deployment, the sampling program generates the output. We discuss each of these programs in what follows.

How a Large Language Model is pre-trained

First, the developer assembles a corpus—an archive of text, including web pages, books, and code. Second, the developer uses the tokenizer to convert that corpus into a sequence of trillions of tokens. Then the developer runs the curve-fitting program, which adjusts the arithmetic function's coefficients by executing the same procedure, over and over, across the whole tokenized corpus. That procedure runs as follows.

1. The curve-fitting program extracts a chunk of the tokenized corpus and submits that whole chunk-sequence to the network.
2. The network executes its arithmetic function and returns its list of probabilities for every position in the chunk-sequence. At each position, the list specifies a probability that each token in the index may appear, computed from the tokens that have appeared up to that position.
3. The loss function computes, at every position in the chunk-sequence, the negative logarithm of the probability the network had assigned to the token that actually followed in the chunk-sequence. Then it averages the resulting quantities across the whole chunk-sequence. The result measures how accurately the network's output—its computed probability distribution—corresponds to the actual token sequence.
4. The curve-fitting program calculates a second list of billions of numbers, one for each coefficient in the arithmetic function. This list is called the gradient. Each number in the list measures how much a small increase in the corresponding coefficient would increase the average loss. The curve-fitting program adjusts each coefficient by subtracting a small multiple of each gradient number from the coefficient to which that number corresponds.

5. Then it extracts the next chunk of tokenized text and executes the whole procedure again.

Each coefficient began at a random value. But after trillions of adjustments, the coefficients have values that make the network's output probabilities approximate the frequencies with which tokens appear together throughout the corpus.

The people who wrote the corpus texts were writing about the world; their words therefore represent their ideas about the world. By contrast, the network's arithmetic function measures patterns among the token sequences that those texts became when the tokenizer converted them. But statistics about patterns among representations do not represent what the representations represented. They represent *patterns among representations.* When the curve-fitting program finishes, the network has obtained no data about the world and no data about what the authors knew. It has obtained data about how tokens were arranged.

How a Large Language Model is post-trained

Pre-training yields a base model that can generate continuations from input text, but this is not yet a chatbot. Developers apply a second stage, called post-training, before deployment. Post-training involves two standard steps, with a third step for "reasoning" models.

The first standard step is called *supervised fine-tuning*. Here the developer assembles a much smaller corpus of example exchanges that human beings wrote or selected, in which a prompt elicits the kind of output the developer wants. The curve-fitting program runs again over this smaller corpus, following the same procedure described above. The coefficients thus shift so that the network assigns higher probability to outputs of the kind that appear in the fine-tuning corpus.

The second step is called *reinforcement learning from human feedback*. Here the procedure goes beyond curve-fitting.

1. The developer has the sampling program generate two or more candidate outputs in response to the same prompt.
2. Human beings rank those candidate outputs against one another. Some developers replace or supplement these human beings with a written constitution of principles and have a language model rank its own outputs against that constitution. The mechanism remains the same either way.
3. The curve-fitting program now executes on the reward model. This second network's arithmetic function receives both a prompt and an output, and returns a single number. The loss function measures how much the reward model falls short of representing the outcome of step 2, which the reward model should reflect by scoring the candidate output that ranked higher above the candidate output(s) that ranked lower. The loss quantity decreases as the gap between the scores widens in the direction their rankings indicate. When the curve-fitting

program finishes, the reward model constitutes an approximate statistical representation of the preferences expressed in step 2.

4. A reinforcement program then has the sampling program generate a new output, passes that output to the reward model for scoring, and adjusts the first network's coefficients to raise the probability of the tokens in outputs that score well with the reward model and lower the probability of the tokens in outputs that score poorly. A penalty term restrains how far the network's probability distribution may move from where supervised fine-tuning had left it.
5. Some developers omit the separate reward model and fit the network directly to the ranked outputs supplied by step 2.

Finally, "reasoning" models add a third step. Where a problem admits an automated check, such as solving a mathematical problem with a known answer or writing code that will either pass or fail certain tests, the developer replaces the reward model with the appropriate check and runs the same optimization against it.

Post-training adjusts which outputs the network favors. It does not change *how* the network generates output. The tokenizer still supplies signs isolated from any reference, and the arithmetic function still returns a probability for each token in the index. Post-training optimization represents human preferences as an additional set of statistics and uses them to calibrate the network's coefficients in ways that may better satisfy users, but still no step in this stage supplies the faculty of interpretation that constitutes *literacy*, that is, *reading comprehension* or indeed *comprehension* at all.

What happens when you use it

1. You submit a prompt.
2. The tokenizer converts your prompt into tokens and passes them to the network as an input sequence.
3. The network executes the arithmetic function and returns its list of probabilities.
4. Just as the curve-fitting program modified the arithmetic function's coefficients during training, now the sampling program samples from the trained arithmetic function to generate a new token. Sampling draws a random token from the index, weighted by the probability the network has computed for it. For example, the sampling program draws a token scored 0.6 six times as often as it draws a token scored 0.1. This randomization is why different instances of the same prompt yield different answers.
5. The sampling program adds each randomly drawn token to the input sequence, then resubmits the sequence, now one token longer, to the arithmetic function.
6. The arithmetic function executes again. It returns a new list of probabilities.
7. The sampling program samples again.
8. And so on. This procedure repeats for every single token.

9. A special “stop” token represents the point where a text ends. The arithmetic function’s output thus includes a probability of stopping. When the sampling program draws that token, it stops sampling, and the tokenizer converts the accumulated tokens into legible output text.

You submit: *I leave ice cream out on the counter. What happens?*

The arithmetic function executes. The sampling program draws *it*, then *melts*, then *(stop)*. The tokenizer converts these tokens into output text: *It melts.*

You know the ice cream would melt, so you consider the answer correct. The sampling program generated it by cycling three times, once for each token, through the procedure outlined above. Nothing in the model furnishes any concept of what ice cream is. The arithmetic function contains coefficients that the curve-fitting program adjusted, through gradient descent, to minimize the quantity that results from the loss function. The human beings who wrote the corpus texts knew what happens to ice cream, their knowledge governed which words they wrote and in which order, and the arithmetic function’s probability distribution has been fitted to statistical tendencies in the sequence of tokens into which their writing was converted. All the LLM ultimately does is calculate probabilities based on how tokens, which are signs severed from any reference, appeared in order in the training corpus.

Why the same model sometimes generates false answers

The arithmetic function computes the probability that tokens appear in a given sequence. It does not compute whether an output is true, whether an output is false, or whether an output contradicts itself or other outputs the model emitted a moment ago.

The same procedure generates both the outputs you know are true and the ones you know are false. Common parlance calls the latter phenomenon a hallucination. That is a misnomer. The mechanism functions as designed; as designed, it cannot distinguish true from false, nor logical from illogical.

A bigger, better corpus does not solve this problem. The curve-fitting program simply minimizes the quantity the loss function measures. That quantity decreases when the network’s computed probabilities move closer to matching the frequencies with which tokens follow each other in the corpus text. In that corpus, “I don’t know” appears infrequently in response to any given question. Thus a network that assigns high probability to generating some attempt at an answer better corresponds to the statistics that represent patterns in its training corpus than one that assigns high probability to generating “I don’t know.” Post-training could correct this problem, but developers grade models on benchmarks that give no credit for an answer of “I don’t know.”[1] Here still, then, a model that generates a bad guess scores better than a model that generates no guess at all.

---

1 Kalai, A. T., Nachum, O., Vempala, S. S., & Zhang, E. (2025). *Why language models hallucinate* (arXiv:2509.04664). arXiv. https://arxiv.org/abs/2509.04664

“Reasoning” models and multi-agent arrangements

The newest models generate a long, preliminary section of output before the final output and present the former section as their reasoning. Developers produce this behavior through the post-training techniques described above. They fine-tune the network’s coefficients on token sequences that place a “problem” text first, a long passage of “reasoning” text next, and a final answer last. Thus the network assigns high probability to generating such “draft” output. Then they run the reinforcement procedure against automated checks, which reward “reasoning” texts that precede passing answers.

In reality, all three stages—the “problem,” the “reasoning,” and the final answer—comprise a single unbroken sequence of tokens, generated one by one through the procedure described above. No separate program intervenes at any point; the reasoning and the answer are one continuous generation. The model does not examine or alter the sequence. The extra tokens improve accuracy on many tasks because those tokens join the input sequence, and the network computes the final output’s probabilities from that longer input. But nothing constrains the final output to agree with the “reasoning,” because no part of the process compares them logically; no logic is applied anywhere in the process; nothing in the model knows what logical agreement is. So it is unsurprising that researchers have shown that a language model’s final output often contradicts the “reasoning” output it prints. However, the same researchers have found that showing users the “reasoning” raises their confidence in the final output regardless of whether the output is, in reality, right or wrong.[2]

A multi-agent arrangement uses yet another program to call the network repeatedly, adding each successive output into the prompt for the next call. Each call to the network executes the same arithmetic function and sampling procedure. Repeating the same statistical computation many times does not amount to logically verifying any output.

What you can inspect

In principle, every coefficient is available to inspect; but reading them does not tell you why the network finally assigns high probability to “melts,” which tokenized texts contribute to that probability, or which of its outputs will turn out to be false. Researchers have identified certain interpretable features in these networks’ internal representations and even traced partial causal accounts for particular outputs, but they have not yet succeeded in producing an account complete enough to tell a user *in real time* why a given output was generated.

---

2 Kambhampati, S., Valmeekam, K., Bhambri, S., Palod, V., Saldyt, L., Stechly, K., Samineni, S. R., Kalwar, D., & Biswas, U. (2026). Position: Stop anthropomorphizing intermediate tokens as reasoning/thinking traces. In *Proceedings of the 43rd International Conference on Machine Learning* (Vol. 306). PMLR.

Generating a single token requires multiplying the input through the entire list of coefficients; each one contributes to the token that results. Each token is thus the product of billions of compounding statistical multiplications. The size of these models, both during the training process that creates them and during their operation, is why they require vast computational resources. This is what drives the current race among LLM vendors to acquire more and more powerful processing chips, more and larger data centers, and the energy to power and cool them.