READ MODE · ANSWER FIRST

LLM Inference Lifecycle

An interactive, visual explanation of LLM Inference Lifecycle, from setup through the complete system.

CHEAT SHEET · 6 ESSENTIAL IDEAS

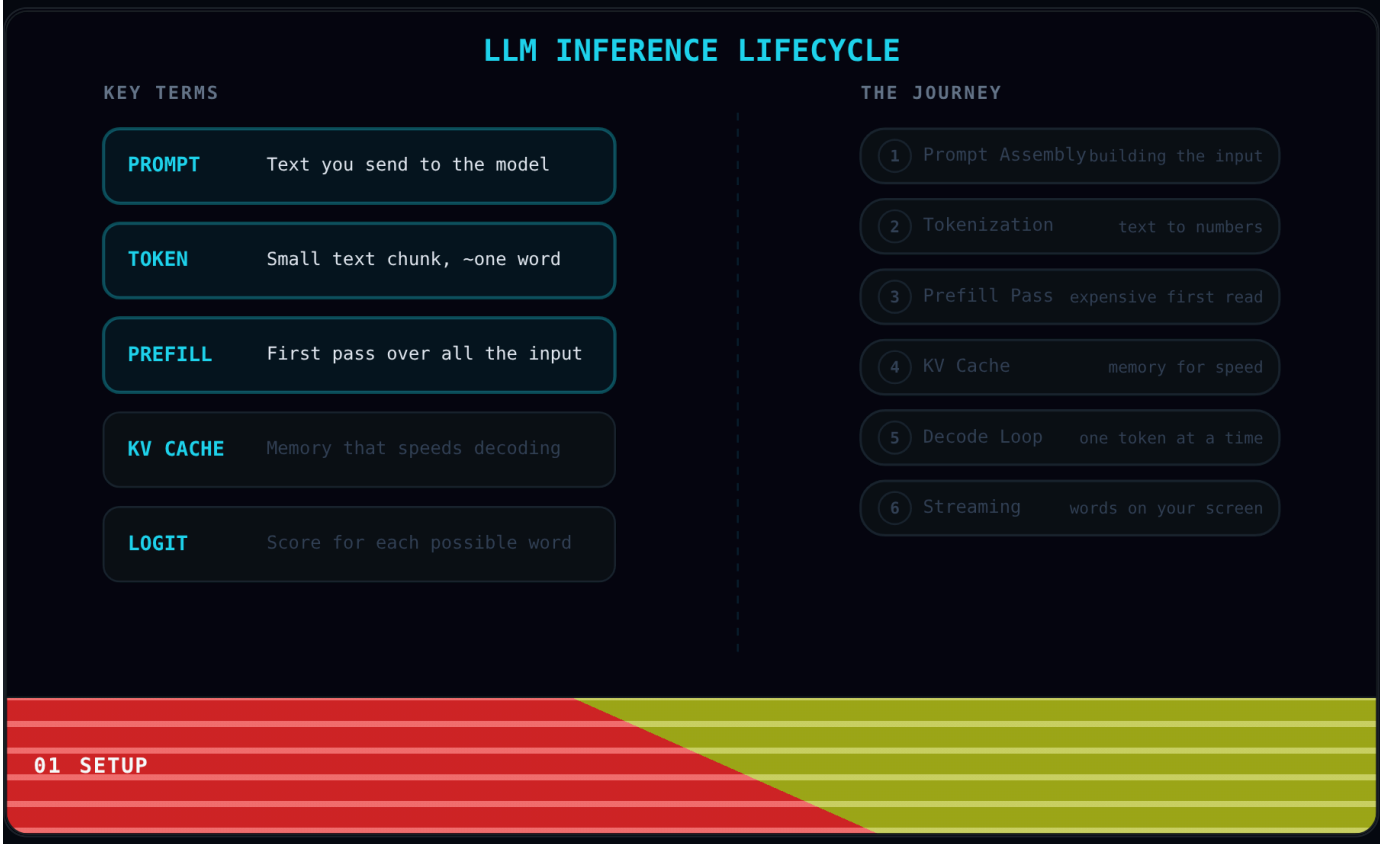
The whole story in 6 lines

LLM Inference Lifecycle becomes easier to reason about when every stage is connected as one system.

1. Show that system text, user turns, and optional retrieval are linearized into one prompt tape before any math begins.
2. Teach that tokenization is a lossy, model-specific segmentation step that produces discrete ids plus special tokens.
3. Contrast prefill with decode by showing all prompt positions moving through layers in parallel.
4. Make the cache feel concrete: a growing memory grid written during prefill, then read during each decode step.
5. Show the tight recurrent loop that appends exactly one token per iteration.
6. Close the story by showing transport packets, buffering, and paint timing on the client side.

STAGE 1 OF 8

Setup



Welcome. This explainer walks through everything that happens between the moment you hit "Send" on a chat message and the moment words start appearing on your screen.

First, some vocabulary. A prompt is the text you send to the model. A token is a small piece of text, roughly one word, that the model processes.

Prefill is the expensive first pass where the model reads your entire prompt at once. The KV cache is the memory that saves that work so the model does not repeat it.

A logit is a raw score the model assigns to every possible next word. Sampling is how it picks one winner from those scores.

Streaming is how decoded tokens travel from the server to your screen, one at a time, instead of waiting for the whole answer.

Now let us begin with the very first step: assembling the prompt.

STAGE 2 OF 8

Prompt

02 PROMPT

The runtime begins with separate sources: system instructions, recent conversation turns, and optional retrieved context. Think of them as ingredients waiting to be combined into a single recipe.

System instructions are placed first. They anchor the model behavior by appearing at the very beginning of the token sequence.

User and assistant turns are appended in chronological order. Try changing the Turns control in the sidebar to see how more conversation history packs into the prompt.

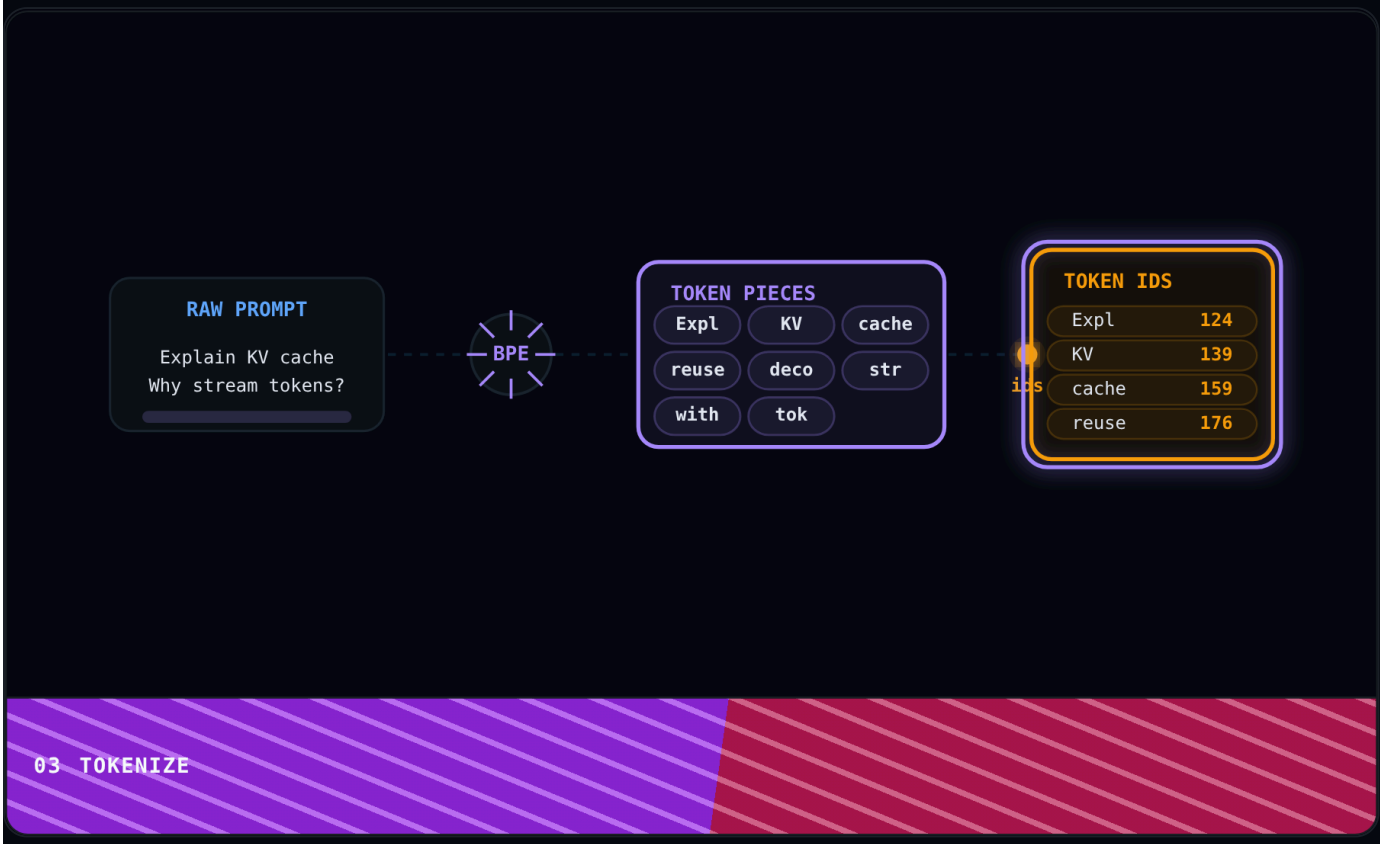
If extra context exists, it is appended as more plain text. Switch the Context control to Lean and notice the prompt shrinks. Switch to RAG or Tools and it grows.

The runtime compacts everything into one ordered prompt tape. Order matters because the model sees position, not labels.

A start marker and the final assistant cursor are inserted. The prompt tape is now ready for the next step: tokenization.

STAGE 3 OF 8

Tokenize



03 TOKENIZE

Now that we have an assembled prompt tape, it needs to be converted into something the model can actually process. Models do not read text. They read numbers.

The tokenizer scans the raw text byte by byte and looks up the best matching pieces from its vocabulary. This is called byte-pair encoding.

Subword pieces appear. Common fragments become reusable tokens. Switch the Vocab control to Code and notice how the same text splits differently.

Each piece is mapped to a stable vocabulary id. The model will only see these integers from this point forward.

Special markers like a start-of-sequence token wrap the sequence. These tell the model where the input begins and where role boundaries are.

The runtime now has one packed integer sequence. Next comes the expensive part: feeding this entire sequence through the transformer stack.

STAGE 4 OF 8

Prefill

Remember how tokenization produced a packed sequence of integer ids? Now every one of those ids is embedded into a dense vector and parked at the entrance of the transformer stack.

The full prompt batch enters the stack together. Unlike the upcoming decode loop, prefill processes all positions in parallel. This is the most GPU-intensive moment of the entire request.

Inside each transformer layer, attention heads fan out across the entire prompt window. Each head learns different relationships between positions.

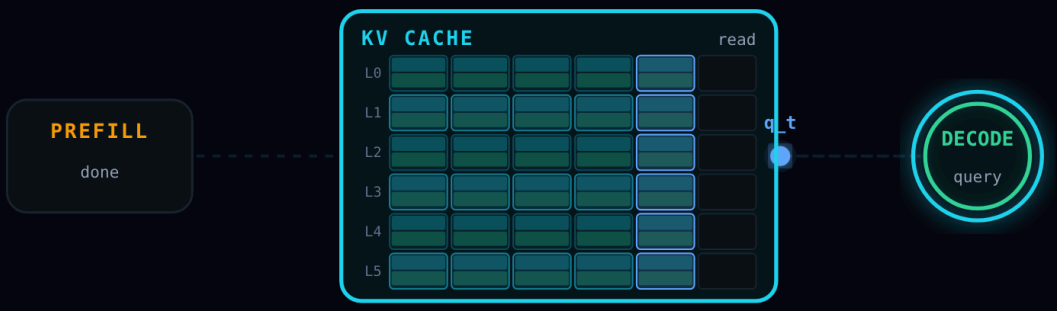
Every layer transforms every prompt position in parallel. Increase the Layers control and watch the grid deepen. More layers means more computation but richer representations.

The last prompt position produces the hidden state that seeds the decode loop. This single vector carries the compressed meaning of the entire prompt.

Along the way, each layer also saved its key and value tensors. Those saved tensors become the KV cache, which is what we will explore next.

STAGE 5 OF 8

KV Cache



05 KV CACHE

We just saw prefill save key and value tensors at every layer. Those tensors land here in the KV cache. Think of it as a spreadsheet: rows are layers, columns are token positions.

Prefill writes keys and values for the entire prompt into cache memory. This is the initial fill that happens once per request.

The cache settles into a reusable grid of prior context. Try changing the Context control to see how more history requires more memory. Switch Policy to Cold to see it start empty.

A new token query arrives from the decode loop. Instead of re-reading the entire prompt through the transformer, it reads the cached keys and values.

Attention reads old cache columns. This is the key insight: the KV cache turns a prompt-length computation into a constant-time lookup per decode step.

After the attention step, one fresh column is appended for the new token. The cache grows by exactly one column per decode iteration. Next, let us see the full decode loop in action.

STAGE 6 OF 8

Decode

Remember the KV cache from the previous stage? The decode loop is what reads from it. Each iteration produces exactly one new token, and that token feeds back as the seed for the next round.

The current last token enters the model alongside the cached keys and values. This is a single-token forward pass, much cheaper than the full prefill.

The model produces logits: a raw score for every word in its vocabulary. Higher scores mean higher probability.

Sampling narrows the candidates. Switch the Sampling control to Greedy and notice only the top bar matters. Switch to Top-p and watch multiple candidates compete.

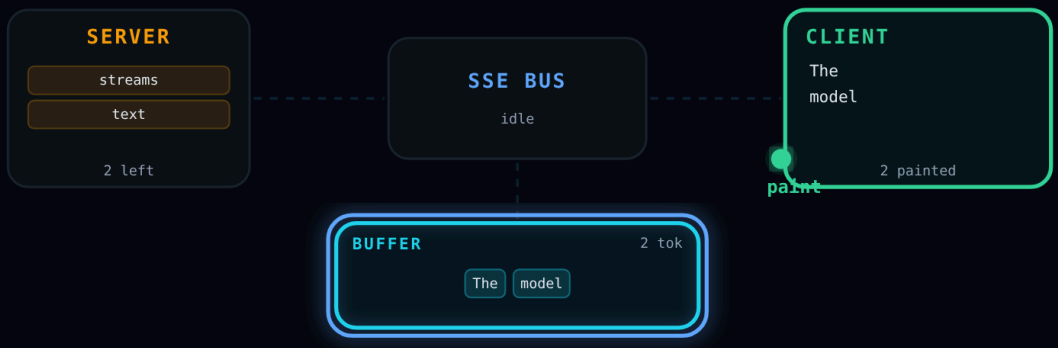
One next token is chosen. With cold temperature, the model is confident and predictable. With hot temperature, it explores more surprising options.

The chosen token is appended to the output sequence and immediately becomes the seed for the next decode loop.

This loop repeats until a stop token or a length limit ends the response. Now let us see how these decoded tokens reach your screen.

STAGE 7 OF 8

Stream



07 STREAM

The decode loop from the previous stage keeps producing tokens. But you do not have to wait for the entire response. Each token can leave the server the moment it is decoded.

The server packages the next token or chunk into a Server-Sent Event and pushes it onto the wire.

Packets travel across the stream transport toward the client. Switch the Transport control to Chunk and notice how multiple tokens are bundled together.

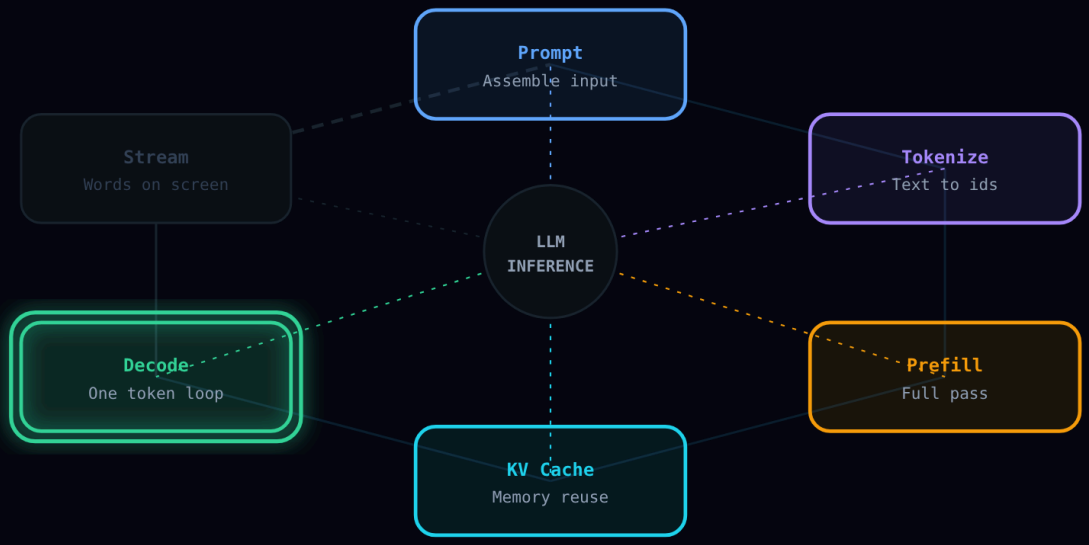
The client runtime receives and buffers arriving events. Switch the Client control to Buffered and notice the client holds tokens before painting.

Visible text is painted into the UI. With live rendering, you see each word appear. With buffered rendering, phrases arrive in bursts.

Streaming continues until a stop token or max-token limit ends the response. Now let us step back and see the whole picture together.

STAGE 8 OF 8

Recap



08 RECAP

We started by assembling a prompt from system instructions, conversation turns, and optional context. That gave us a single ordered text tape.

We then tokenized that text into integer ids. The model never sees raw text. It only sees numbers.

Prefill processed the entire token sequence through every transformer layer in one expensive parallel pass.

That pass saved keys and values into the KV cache, a memory structure that lets future tokens skip recomputation.

The decode loop then ran one token at a time: reading the cache, producing logits, sampling a winner, and feeding it back.

Finally, streaming delivered each decoded token to your screen incrementally, so you did not have to wait for the full response.

That is the complete LLM inference lifecycle. From prompt assembly to streaming output, every step is visible in what you just walked through.