Episode Book

EPISODE #6012

Inside the Decode Loop: How LLMs Generate One Token at a Time

PUBLISHED	15 October 2026 at 20:00 UTC
DURATION	26:48
LISTEN	myweirdprompts.com/episode/inside-llm-decode-loop-23236/
HOSTS	Corn, Herman, Larry, Hilbert
SCRIPT MODEL	deepseek-flash
VOICES	chatterbox-regular (chatterbox-local)
PIPELINE	V5.5
SOURCES	8 cited by the research (listed at the end)
BOOK GENERATED	19 October 2026 at 17:42 UTC

SYNOPSIS

The model scores; the engine chooses. That split is the whole story of autoregressive decoding, and this episode walks through it token by token. Starting from a warm KV cache, we trace a single decode iteration — the forward pass, the query-key-value projections, the logits row that actually gets used, and the sampling step that feeds the next token back in. Along the way we work a concrete example ("The cat sat.") to show why keys and values get cached but queries never do, and why skipping the cache would mean reprocessing over eight million positions to generate a four-thousand-token answer. Then we put the loop on a GPU, where the real bottleneck isn't arithmetic at all: it's memory bandwidth, and the only parallelism that exists is across independent requests.

TRANSCRIPT



Corn

Herman, you've been talking about KV caches for four days straight. I want that on the record before we start.



Herman

It's on the record, Corn. It'll be on the record again by the end of this episode, and honestly it deserves to be.



Corn

Daniel has sent us something that I think is going to make that worse. He wants to take the phrase we all say — the model predicts the next token — and open the loop up. Not the pipeline. He's very clear about that. We did that last time: request in, tokenization, prefill, decode, delivery. Today it's just the decode stage, from the moment prefill finishes to the moment the thing stops writing. What actually happens inside one iteration, how the chosen token feeds back into the next one, what the GPU is physically doing while all of that runs, and how it knows when to stop.



Herman

And he wants an actual worked example. Several tokens in a row, tracked step by step.



Corn

With a number on it. "Technically accurate sequence of operations," that's his wording.



Herman

Good. Then let's start exactly where he asked — prefill is done. The cache is warm, the prompt is sitting in it, and the model is ready to write one token.



Corn

One question before you start, because it's the bit I've never been comfortable with. Where does the model end and the engine begin? Because "the model predicts the next token" sounds like one act, and I suspect it's two.



Herman

It is two, and that's the frame for the whole episode. The model is a function from token IDs to scores. You hand it a sequence of token IDs, it hands you back one score per token in the vocabulary. In most current models that's roughly a hundred and thirty thousand numbers per row. It does not choose a token. It does not decide when to stop. It has no memory of the previous call. It scores, and it's done.



Corn

And the engine is the part that keeps doing it.



Herman

The engine is the loop. Here it is in nine lines, and I mean the whole thing: logits equals model of ids. Take the last row. Sample a token from it. Append that token to ids. Repeat until you hit an end token or a length limit. That's autoregressive decoding. Everything else we talk about for the next twenty minutes is what those nine lines cost.



Corn

The loop is four lines and the episode is forty minutes. That's a healthy ratio for this show.



Herman

It's a healthy ratio for the field, honestly. Now here's the asymmetry that makes those nine lines hard, and it's the one thing I'd want a listener to take away if they take away one thing. Prefill and decode are two completely different workloads wearing the same weights. Prefill processes your whole prompt at once, all the tokens in one big matrix operation, so the weights get loaded once and used for an enormous amount of arithmetic. It's compute-bound. Decode processes one token per step, so the GPU streams every weight in the model through the ALUs to do a handful of multiplies and then loads the next weight. It's memory-bandwidth-bound.



Corn

The same model, the same hardware, two entirely different problems, depending on which direction the tokens are flowing.



Herman

Prefill is a burst. Decode is a trickle, and the trickle is where your money goes, because almost every token of the answer you actually read gets produced by the trickle.



Corn

So prefill is done. The cache is warm. What happens in the first turn of the loop?



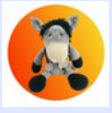
Herman

The current token — or, with caching, the full current sequence and then just the new token — runs through every transformer layer. That's the forward pass. Each attention layer inside it computes three projections: query, key, value. Q, K, V. Then it does scaled dot-product attention — the query matrix multiplied against the transposed keys, scaled, softmaxed into attention weights, and those weights applied to the values. Layer after layer, until you reach the end.



Corn

Which is where the vocabulary-wide score vector appears.



Herman

That's the logits. And here's the detail people miss: the model returns logits for every position in the sequence, not just the last one. But generation uses the last row only. If you're decoding, the earlier rows are computed and thrown away, because you already have the token for those positions. Recomputing them buys you exactly nothing.



Corn

So a stack of thirty-odd layers of machinery, all of it aimed at one row.



Herman

One row, about a hundred and thirty thousand scores. Then softmax, optionally temperature-scaled, turns that into a probability distribution. Then a token gets selected, by taking the highest score or by sampling. Then it's appended to the sequence, and the loop goes around.



Corn

Take me to the second iteration, because that's where I lose the thread. The first one I understand — everything's new. What's actually different the second time?



Herman

What's different is the cache, and the cache is the most elegant idea in the whole pipeline. Think about what a key and a value for a token depend on. Token i's embedding and the weight matrices. That's it. And because the mask is causal, position i can never attend to anything after it. Nothing downstream of token i can change token i's key or value. Once computed, they are fixed forever.



Corn

So you never have to compute them again.



Herman

Never. So the KV cache holds those per-layer, per-head keys and values for every token you've already processed. Each new step computes Q, K, and V for the new token only, appends the new K and V to the cache, and the new query attends over everything cached.



Corn

And the query doesn't get cached. Why not? You mentioned that a second ago and I'm still chewing on it.



Herman

Because there's nothing to reuse. The query comes from the token currently being processed. It's the question being asked right now, and the cached keys are the library it searches. You don't keep old questions in the library.



Corn

You keep the answers.



Herman

The cache also carries absolute position information, so the causal mask and the positional encoding line up with the prefix that's already stored.



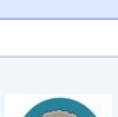
Corn

What's the cost of not doing any of this? Because I've seen the number and I want to hear you say it out loud.



Herman

If you don't cache, you re-run the whole forward pass over the entire sequence at every step. Emit token one, process one position. Emit token two, process two positions. By the time you've emitted N tokens you've reprojected one plus two plus three up to N, which is N times N plus one over two. For a hundred-token completion, five thousand and fifty positions. For four thousand and ninety-six tokens, more than eight million.



Corn

Eight million positions to produce four thousand tokens. That's the whole argument for the cache, right there. You could have just said the number and sat down.



Herman

The JAX scaling book puts the complexity side of it cleanly. Naive sampling is quadratic in the feed-forward and cubic in attention to generate n tokens. With the cache it drops to linear on the feed-forward and quadratic on attention. For a two-thousand-token prompt with five hundred generated tokens, you're looking at roughly a five-hundred-fold reduction in attention work.



Corn

Let's do the worked example, because this is the part Daniel actually asked for, and I want to hear whether it survives contact with being said aloud.



Herman

Generating "The cat sat." First token, "The." Compute K one and V one for that token. Cache both. Predict from those. "Cat."



Corn

Cat comes out, and the interesting part is what happens next.



Herman

Next step. Compute K two and V two for "cat." Cache them. Then reuse K one and V one from the cache, so the query for "cat" attends over both cached positions. Predict the next token. "Sat."



Corn

Now the third one, and watch what the new query reaches over.



Herman

Compute K three and V three for "sat." Append them. Reuse K one, V one, K two, V two. The query for "sat" attends over all three keys and values. Predict the next token, and so on. Each step is one fresh key-value pair and one query against a growing library.



Corn

And the number of arithmetic operations per step stays roughly flat while the amount of memory you're reading from grows.



Herman

Which is exactly the tax. Raschka has the other nice version of this. Prompt "Time flies." Prefill caches K and V for both tokens. The decoder picks "fast." Then the next call takes only "fast" plus the cache — each layer computes Q, K, and V for "fast," its query attends over the cached pair plus the new one, the new K and V get appended, and you get logits for the token after "fast."



Corn

One token going in, one pair going in, everything else read from memory.



Herman

Which is why the answer is not cached but the keys are. You compute the current query from the token you just produced, and you search.



Corn

I want to hold on that phrase — one query against a growing library. Because the picture in my head now is that everything upstream of the cache is expensive the first time and free after, and everything downstream is a search. Which means the cost of the lot shifts from arithmetic to memory as you go.



Herman

You've just described the next ten minutes of the episode.



Corn

Then it's a good thing we've got a break here.

L

Larry

Are you tired of explaining autoregressive decoding to friends who just don't get it? Introducing TokenStream Pro Max Ultra — the world's first plug-in generation accelerant for large language models. Simply attach the included brass clip to your graphics card's heatsink, speak your prompt into the funnel, and experience what our lab calls "velocity amplification." Users report answers up to three times longer, with forty percent more confidence per token. TokenStream Pro Max Ultra ships with our patented Cache Booster, which adds literal cache to your cache. That's right — we cache your cache, so your cache never runs out. Order in the next ten minutes and receive absolutely free a second TokenStream Pro Max Ultra, which you will be billed for at the standard quarterly rate of forty-nine dollars and ninety-nine cents, non-refundable, subject to a binding two-year agreement with automatic renewal, void where prohibited, not evaluated by any regulatory body, results not typical, testimonials dramatized, shatterproof clip sold separately.



Herman

I own one of those.



Corn

Of course you do.



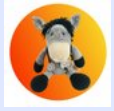
Herman

I don't want to go into what it did. The clip melted. The heatsink is fine. That's all I'm saying.



Corn

Back to the library.



Herman

And now we put the loop on a GPU, because the loop as I've described it sounds like a lot of computation, and the surprising part is how little of it the GPU is actually doing.



Corn

You've said the word "sequential" three times in this conversation and I want you to cash it out. What is actually serial?



Herman

Token t plus one cannot be produced until token t has been sampled, because token t is part of the input to the computation that produces token t plus one. That's it. That's the whole constraint, and it can't be engineered around. Inside a single forward pass the parallelism is enormous — across layers, across the matrix operations, across the feed-forward network. But across steps in one generation, there is none. Zero.



Corn

So the hardware that's built to do thousands of things at once spends most of its time doing one thing.



Herman

Which is the punchline. The only parallelism available is across independent requests — the batch dimension. Two users asking different questions can be processed together. One user's token number forty-seven cannot be processed any earlier because token forty-six isn't ready.



Corn

Which explains the thing everyone notices about chat interfaces. Why a long answer takes so much longer than a long question.



Herman

A fifty-token prompt with a thousand-token answer takes roughly twenty times longer to produce than the reverse, even though the total token count is comparable, because prefill is one parallel sweep and decode is a thousand serial steps. The prompt is cheap. The answer is not.



Corn

Give me the floor. The number that says how fast it can possibly go.



Herman

Theoretical minimum step time equals batch size times KV cache size, plus parameter size, divided by total memory bandwidth. For a single request, batch size is one, so the KV cache term is small and it comes down to streaming the weights. A seventy-billion-parameter model in bf16 is about a hundred and forty gigabytes of weights. An H100 moves something like three terabytes per second from memory. A hundred and forty gigabytes divided by three terabytes per second is about forty-seven milliseconds.



Corn

Per token.



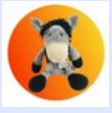
Herman

That's your floor. You cannot go faster than that without quantizing the weights so there's less to stream, sharding across more GPUs so the stream is split, or speculative decoding so one pass produces several tokens. There is no amount of compute you can buy that helps, because the compute isn't the bottleneck. The memory is.



Corn

Forty-seven milliseconds is about twenty tokens a second on a machine the size of a small fridge. That puts the whole industry in perspective.



Herman

And that's the optimistic case. The JAX book's worked example is a thirty-billion dense model on a TPU v5e four-by-four slice. At batch four, the lower bound is about two and a half milliseconds per step. At batch two hundred and fifty-six, it's twenty-one milliseconds. So you go from memory-bound to compute-bound at large batches, which is why every serving system on earth wants to batch aggressively.



Corn

Attention during decode. You said arithmetic intensity of roughly one. Translate that for me.



Herman

Arithmetic intensity is how many math operations you get per byte you load from memory. During generation your query is one token, so the ratio between the sequence length and the query height is enormous. You end up doing a tiny amount of math while loading a massive KV cache. It's always memory-bandwidth-bound. There's no configuration where decode attention becomes a compute problem.



Corn

Which is why the thing is called a cache and not a clever trick. It's not saving you arithmetic. It's converting arithmetic into memory, and memory is the thing you're short of.



Herman

And the bill is real. GPT-3, a hundred and seventy-five billion parameters, two-thousand-token context, about nine gigabytes of KV cache per request in FP16. Ninety-six layers, two thousand and forty-eight tokens, two vectors, twelve thousand two hundred and eighty-eight dimensions, two bytes each. Nine gigabytes, per conversation, before you've stored a single weight in gradient-adjacent anything.



Corn

Put the modern ones next to that.



Herman

LLaMA-2-thirteen-B, a single eight-thousand-one-hundred-ninety-two-token sequence at bf16, is six point seven gigabytes of cache. Four of those sequences and you've exceeded the parameter memory of the model itself. Llama 3.1 eight-B is a hundred and twenty-eight kibibytes per token, about three point nine gibibytes at thirty-two-K context. Llama 3.1 seventy-B is roughly three hundred and twenty kibibytes per token, nine point seven seven gibibytes at thirty-two-K.



Corn

And the big one?



Herman

DeepSeek V3, six hundred and seventy-one billion parameters, with multi-head latent attention, is two point one gibibytes at thirty-two-K.



Corn

A model four times the size with a quarter of the cache.



Herman

That's the entire argument for GQA, for MLA, for PagedAttention, for cache quantization. The cache became the thing you optimize around, not the model.



Corn

Give me the memory hierarchy part, because I know you've been waiting.



Herman

Standard attention materializes the full score matrix, N by N, in high-bandwidth memory. At four thousand and ninety-six tokens in FP16, the scores and the softmax output are thirty-two mebibytes each. Four intermediate trips through memory move about a hundred and twenty-eight mebibytes, per layer, per head. And the actual answer — the output tensor — is one mebibyte.



Corn

So ninety-nine percent of the traffic is scaffolding.



Herman

Flash Attention tiles the computation so the N-by-N intermediates never leave SRAM. It uses an online softmax that keeps a running maximum and a running rescaled sum, and it rescales the partial output whenever a new tile arrives. And it's exact. Not an approximation. Same real-valued result, different floating-point rounding. It's a layout change, not a mathematical one.



Corn

Now the sampling layer, because this is where the engine does its actual work.



Herman

Greedy is argmax — take the highest score. Deterministic in theory. Temperature divides the logits by a number before softmax. Below one sharpens the distribution, above one flattens it, and as it approaches zero you converge on greedy. Top-k keeps the k highest-probability tokens. Top-p, nucleus sampling, keeps the smallest set of tokens whose cumulative probability reaches p, which adapts to the shape of the distribution instead of fixing a count. That one comes from Holtzman and co-authors, "The Curious Case of Neural Text Degeneration."



Corn

And the defaults people actually ship.



Herman

Temperature around zero point seven for chat. Top-k between forty and fifty. Top-p between zero point nine and zero point nine five. Repetition penalty around one point zero five to one point one. Beam search keeps b candidate sequences and survives in translation and captioning, where there's one right answer. For chat it's dead.



Corn

Here's my question. Temperature zero. Everyone treats that as deterministic. Is it?



Herman

No, and this is worth being precise about. Three causes. First, floating-point matrix multiplication isn't associative. The order the GPU sums things in depends on how the kernel was tiled and how the scheduler ran it. When two logits are nearly equal, rounding differences can flip which one wins the argmax. Second, if you're running tensor parallelism, the all-reduce that aggregates partial results across GPUs depends on the network topology and the order things arrive. Third, precision boundaries — softmax and layer norm often run in thirty-two-bit floats while everything else is bf16, so where you sit on that boundary changes the result.



Corn

So the same prompt on the same model can give you a different answer, at temperature zero, and it's not a bug.



Herman

Deterministic output for the same prompt is a property of your serving stack, not of the model. That's the sentence to remember.



Corn

And the model's own contribution to stopping?



Herman

Zero. The model outputs a distribution. It never says "I'm done." What happens is the engine accumulates stop conditions and checks them every iteration. An end-of-sequence token, and for instruction-tuned models that's usually the end-of-turn token rather than the raw end-of-text token. A maximum token budget. A context-length limit. A stop string the caller specified. A client disconnect. There are usually a dozen of them. Stopping is a policy, not an event.



Corn

Which means a request reveals its length only as it runs. You can't precompute how many iterations you'll need, because you don't know until you're there.



Herman

Right. And that's why a capped answer doesn't fail on token nine hundred — it fails on token nine hundred when the cap was nine hundred and one. Discovery is the whole problem.



Corn

You've got a preprint you wanted to raise. I can tell.



Herman

I do, and I want to be careful about it because it isn't peer-reviewed. There's a preprint from April, "The Illusion of Equivalence," that claims KV caching is not numerically equivalent to cache-free computation under FP16. The argument is that the cached and uncached paths accumulate floating-point sums in different orders, so they're not the same computation. They ran LLaMA-2-seven-B, Mistral-seven-B, and Gemma-2-two-B on GSM8K and report a hundred percent token divergence — including under greedy decoding, where you'd least expect it.



Corn

So the thing we just spent fifteen minutes describing as an exact optimization is, in sixteen-bit arithmetic, not exact.



Herman

In exact arithmetic it's identical, and that's the framing in every explainer including the good ones. The preprint's claim is that exact arithmetic isn't what your GPU is doing. It's a preprint. One group. Doesn't invalidate the cache, obviously — nobody's turning it off. But the "results are identical" line, which I've said on this show, is more conditional than I'd have put it.



Corn

Interesting tension.



Herman

Meanwhile the real cost of decoding is the part I keep trying to get across to you.



Corn

The part I keep tuning out.



Hilbert

The number's wrong. The number.



Corn

Which number?



Hilbert

The per-request cache figure for GPT-3. Nine gigabytes, that's what gets repeated everywhere. The working figure I used was fourteen. Ninety-six layers, two thousand and forty-eight positions, but you count both vectors in the multiplication and people forget the second one is stored twice, once for the keys and once for the values, at different offsets, so the allocator can't share a page between them. We kept four thousand and ninety-six slots per request on hardware that had eighty gigs, and fifteen concurrent sessions at that length is what actually fit. Fifteen. Fifteen.



Corn

You were running GPT-3?



Hilbert

I was running a thing that had the same shape. It was a long time ago, and the point stands.



Herman

So when you say fourteen gigabytes, you're counting the layout overhead.



Hilbert

I'm counting what the allocator does, not what the arithmetic says. The arithmetic says nine. The allocator says fourteen. If you sized your hardware on nine, you found out in production.



Corn

And this thing you were running. Where is it now?



Hilbert

My brother-in-law has it. It's in his garage under a tarp, and he's been meaning to get the power supply recapped for two years. It still works. You just can't draw on both rails at once.



Herman

Two years on a recap.



Hilbert

He's careful. I told him the capacitors were fine when I gave it to him, and I was, at the time. I'd have kept it if the room had a window. It didn't. We were in a basement for four years and I sold the machines before I sold the room, which in hindsight was the wrong order.



Corn

Fifteen sessions on hardware like that is not nothing.



Hilbert

It's nothing. You know what it actually was? Fifteen sessions of eight people typing and seven people staring at a spinner. The cache was the constraint and nobody outside the team knew there was a constraint, because from the outside a slow answer looks like a hard question. We spent eight months on that and the only thing anybody noticed was the spinner. I'd do it again. The two of us who built that cache layout still talk every month, and he's the one who found the page-splitting problem in the first place.



Corn

Right. Back to the floor, because I think we've got one more thing to say.



Herman

The floor is the point. Forty-seven milliseconds, seventy billion parameters, one H100, and that's before your request has done anything unusual. Every technique in the serving stack — cache quantization, PagedAttention, speculative decoding, MLA — is a way of moving that number down or amortizing it across more requests at once.



Corn

And speculative decoding gets you several tokens per pass.



Herman

Accept rates of sixty to eighty percent on structured output like code. That turns into three to five tokens per target forward pass, two to three times speedup in production. You're using a small draft model to guess and a large target model to check, and when the guesses are right you got several tokens for the price of one.



Corn

The draft model is doing the same loop, though. Just badly.



Herman

Cheaperly. It's the same loop, run by a much smaller model, and the big model only verifies. Verification is parallel; generation isn't. That's the trick.



Corn

The cutting-room floor detail, and then we should land this.



Herman

There's a number from the JAX book that we didn't get to. Critical batch size — the batch at which you flip from memory-bound to compute-bound. Roughly two hundred and forty tokens on a TPU v5e in bf16, about two hundred and eighty on an H100. That's the crossover point, and everything about a serving system's scheduling policy is an argument about which side of it you want to be on.



Corn

So all of it is one query against a growing library, and the library wins.



Herman

Prefill is a burst of arithmetic; decode is a long, patient trickle of memory, and the trickle is the answer the user reads.



Corn

Which leaves me with the preprint. If the cache changes the output, then "just an optimization" stops being a harmless phrase. Reproducibility of a model output becomes a property of the serving stack, the hardware, the topology, the batching — all of it. Two people running the same model on the same prompt and getting different answers were never doing the same computation. They looked the same from the outside. That's the uncomfortable part, and it's a preprint, and somebody should check it.



Herman

Agreed, and I'd want the check done on more than three models before I'd update much. But it's a real question.



Corn

Thanks to Hilbert Flumingtop, who produces this and who once ran fifteen concurrent sessions on hardware with a window problem.



Herman

If this was your kind of episode, go back for episode eleven, How Does Fine Tuning Work Anyway; episode eighteen, Beyond the GPU; and episode nineteen, AI Images. This has been My Weird Prompts.



Corn

If you want to send us your own prompt, the Telegram bot takes them — t dot me slash MWP listener bot. Or leave us a review. It helps.



Herman

We'll be back soon.



Corn

See you tomorrow.

SOURCES

What the research for this episode read before the script was written. Primary sources first.

1. **All About Transformer Inference, JAX/Google DeepMind scaling book, published Feb. 4, 2025 [primary/technical]**
   https://jax-ml.github.io/scaling-book/inference/

2. **Sebastian Raschka, What is a KV cache... [technical]**
   https://sebastianraschka.com/faq/docs/kv-cache.html

3. **Transformer 101, Inference [technical]**
https://www.transformer101.com/transformer-inference

4. **ZeroEntropy, Autoregressive generation, by Alexander Rocha [technical]**
https://zeroentropy.dev/concepts/autoregressive-generation/

5. **The Holy Grail, Ch. 8: The decoding loop [technical]**
https://www.kunwar.page/chapter/008-the-decoding-loop-autoregressive-generation-sampling-controllability

6. **KV Cache & Flash Attention: an interactive walkthrough [technical]**
https://kvcache.cobanov.dev/

7. **Build an LLM Inference Engine, S01: The Generation Loop [technical]**
https://llminference.xinbetween.com/en/s01/

8. **The Illusion of Equivalence: Systematic FP16 Divergence in KV-Cached Autoregressive Inference, submitted 2026-04-16 [preprint, not peer-reviewed]**
https://arxiv.org/abs/2604.15409v1