Episode Book

EPISODE #6011

Inside LLM Inference: Prefill, Decode, and the KV Cache

PUBLISHED	15 October 2026 at 19:59 UTC
DURATION	34:39
LISTEN	myweirdprompts.com/episode/llm-inference-prefill-decode-kv-cache-34131/
HOSTS	Corn, Herman, Larry, Hilbert
SCRIPT MODEL	deepseek-flash
VOICES	chatterbox-regular (chatterbox-local)
PIPELINE	V5.5
SOURCES	15 cited by the research (listed at the end)
BOOK GENERATED	19 October 2026 at 17:42 UTC

SYNOPSIS

A single request to an LLM isn't one computation—it's a stack of stages wearing a trench coat. This episode walks the full pipeline: flattening a chat transcript into one flat document with special role tokens, tokenizing text into IDs, the massively parallel prefill pass that produces your first token, and the strictly sequential decode loop that produces every token after. Along the way: why forgetting one flag called add_generation_prompt makes the model write your half of the conversation, why "encoding" and "decoding" each mean two different things, and how a KV cache can quietly outgrow the model weights it's serving. If you've ever wondered where your time-to-first-token actually goes, this is the tour.

TRANSCRIPT



Corn

Here's a number that shouldn't be legal. One sequence, eight thousand tokens, on a thirteen billion parameter model. The key-value cache for that single sequence is six point seven gigabytes.



Herman

And the weights are only twenty-six.



Corn

Right. Four sequences of context and you've spent more memory holding the conversation than holding the model.



Herman

That's the thing Daniel's poking at, isn't it. He said he's been thinking of inference as one discrete event. Request in, model does its work, response out.



Corn

That's how we all talk about it.



Herman

It's how the pricing pages talk about it.



Corn

It's how I talk about it, and I know better. So he wants to open the engine up. Assemble the prompt, tokenize it, prefill, decode token by token, detokenize, hand it back. And then he asks the question underneath all of that, which is whether any of it is one computation or whether it's a stack of unrelated ones wearing a trench coat.



Herman

He also wants the words sorted out. Encode, decode, prefill, tokenize. He's noticed those get used for different things depending on who's talking.



Corn

Which they do, and it's confusing, and I'll defend anyone who's muddled by it.



Herman

Daniel's framing has five stages. Request prep, tokenization, prefill, decode, detokenize and deliver.



Corn

There's a sixth one hiding in there and I'll get to it. But five is a fair map.



Herman

Before we start walking it, one thing worth saying about the word inference itself. In machine learning generally, inference is applying a trained model to production data to produce an output. A prediction, a classification, a score, an embedding.



Corn

And the moment you're serving an LLM, the word swallows the whole request lifecycle. Queue, schedule, prompt processing, generation, delivery.



Herman

Most of which is not neural network computation at all.



Corn

Say that part again, because it's the part people miss.



Herman

Converting text into tokens and tokens back into text requires no neural networks. A tokenizer is a lookup table with opinions. A mapping between strings and numbers.



Corn

So the phrase LLM inference covers a text-processing step, a scheduling problem, and a matrix multiplication problem, and we use one word for all three.



Herman

That's why the pipeline is worth walking. Because once you see the stages laid out, "inference" stops meaning anything specific and starts meaning "the whole shift."



Corn

Start at the beginning. The model never sees a chat window.



Herman

It sees text. One flat document.



Corn

So somebody has to flatten the chat into it.



Herman

That's request preparation. System prompt, user messages, assistant replies, tool calls, tool results. All of it gets concatenated into one stream with role markers separating the pieces. Then the model does the only thing it knows how to do, which is continue the text.



Corn

Continue from where.



Herman

From right after a marker that says the assistant is about to speak. That's the whole trick. You don't ask the model a question, you write a transcript in which the answer hasn't happened yet and you let it finish the document.



Corn

The markers aren't words, are they.



Herman

No, and this is the part that surprises people. They're special tokens. Separate entries in the vocabulary with their own IDs. They're not the characters for a less-than sign and a pipe. They're single units the tokenizer recognizes as one thing.



Corn

And every model family has invented its own.



Herman

Qwen uses ChatML. User turn opens with the im start marker, closes with im end. Llama 3 uses start header id and end header id, and the turn ends with the eot id token. Mistral uses square bracket INST for instructions. gpt-oss uses something called harmony.



Corn

So the same conversation typed into two different models gets wrapped two different ways.



Herman

And you can't use the wrong wrapping. Llama 3's template has to be applied to Llama 3. In Hugging Face the template ships as a Jinja template alongside the tokenizer, and you call it with your list of messages.



Corn

There's a flag in that call that matters more than it should.



Herman

Add generation prompt. It appends the marker that says the assistant is about to speak. Leave it off and the model doesn't answer you.



Corn

What does it do.



Herman

It carries on writing your message. You asked it a question, the transcript ends mid-thought, and the most likely continuation is more of you talking. It'll invent your next paragraph.



Corn

That's a wonderful failure. You forget one flag and the model writes your half of the conversation and looks confident about it.



Herman

And it's not a bug in the model. It's doing exactly what you told it to. You just told it to keep going.



Corn

So the transcript is assembled. Then what.



Herman

Tokenization. The text gets split into tokens, and each token maps to an ID from a fixed vocabulary. Whole example, plain sentence. The quick brown fox jumps over the lazy dog.



Corn

Classic.



Herman

It becomes ten tokens. The, space quick, space brown, space fox, and so on. And each one looks up to a number. The gets nine hundred and seventy-six. Quick becomes four thousand eight hundred and fifty-three. Lazy becomes twenty-nine thousand and eighty-two.



Corn

And the model only ever sees the numbers.



Herman

Only the numbers. And the vocabulary is fixed before training ever starts, so the split into tokens is decided by the tokenizer, not by the model. The model has no say in how the text got chopped.



Corn

Which means the chopping can go badly.



Herman

Constantly. That's why tokenizers are trained on a corpus before the model is. They learn which chunks of text show up together often enough to deserve their own entry.



Corn

Give me the algorithms.



Herman

Three that matter. Byte pair encoding, which is what Llama, Gemma and Qwen2 use. Unigram, which is T5 and Pegasus. And WordPiece, which is the BERT family.



Corn

How does byte pair encoding work, in one pass.



Herman

Start with single characters. Count every adjacent pair across the corpus. Merge the most frequent pair into a new token. Repeat forty thousand times.



Corn

That's it?



Herman

The original GPT vocabulary is forty thousand four hundred and seventy-eight entries. Four hundred and seventy-eight base tokens and forty thousand merges. GPT-2 came to fifty thousand two hundred and fifty-seven. And most modern models are over a hundred thousand.



Corn

The numbers keep growing.



Herman

Vocabulary is cheap relative to model size and it buys you shorter sequences. Short sequences are everything in this business.



Corn

Now the terminology trap, because it's worth flagging here.



Herman

There are two meanings of encoding and two of decoding. Mistral's own documentation calls the text-to-IDs direction encoding and the IDs-to-text direction decoding.



Corn

Fine. But encoding also names the encoder stack in an encoder-decoder model.



Herman

And decoding also names autoregressive generation. So a sentence like "the encoder encodes the prompt and the decoder decodes the output" uses both pairs of meanings in the same breath.



Corn

Which is why people who know what they're talking about still sound confused.



Herman

Keep them separate. Tokenizer encoding is a lookup. Model encoding is a stack of layers. Say which one you mean.



Corn

Right. The tokenizer sits outside the transformer entirely.



Herman

Before the model runs and after it finishes. It's the door on both sides.



Corn

So we have numbers. What happens to the numbers.



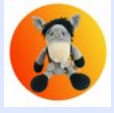
Herman

Prefill.



Corn

Say what it is.



Herman

The entire prompt goes through the model in a single forward pass. All the layers, all the tokens at once. Every token gets a query, a key and a value computed for it. The keys and values get stored. And at the end you get logits for the first output token.



Corn

Logits being one raw score per vocabulary entry.



Herman

One per entry. A hundred thousand numbers, and the highest one wins whatever comes next.



Corn

So the entire prompt is processed before a single word comes out.



Herman

Every token of it. And that's why prompt processing dominates the wait on a long input.



Corn

This is the stage where the parallelism question actually has an answer.



Herman

It does. Because the whole prompt is known up front, the prompt positions can be computed in parallel inside a layer. Under the attention mask, token nine hundred doesn't have to wait for token eight hundred and ninety-nine to finish.



Corn

Layers still go in order.



Herman

Layers always go in order. Layer two needs layer one's output. You can parallelize across positions, never across depth. That distinction runs through this whole episode.



Corn

And the hardware profile of prefill.



Herman

Compute-bound. NVIDIA's description is that it's a matrix-matrix operation, highly parallelized, and it effectively saturates GPU utilization. You're feeding the card big dense blocks of arithmetic and it's working flat out.



Corn

The metric people quote for it.



Herman

Time to first token. How long between sending the request and seeing the first character appear.



Corn

And that's where the money goes on a long prompt.



Herman

That's where it goes. Prefill scales with the length of the input. Double the prompt, roughly double the work.



Corn

So the pipeline so far is flatten the chat, chop it into tokens, look up the numbers, then one big parallel pass over the whole thing.



Herman

And out the other end, one token.



Corn

One token after all of that.



Herman

Then you do it again.

L

Larry

Are you still hosting inference like it's nineteen ninety-eight? Tired of paying per prompt? Introducing PromptSlurry, the world's first liquid prompt delivery system. Pour one scoop into your server rack and every model on the floor gets faster. Handles context windows up to the length of a municipal swimming pool. Works on GPUs, TPUs, and, frankly, any warm object with a fan in it. Normally four hundred dollars. Now three hundred. Now two hundred. Now ninety-nine. Now it's free. Now we're paying you. Throw in a lifetime supply of token brackets, a decorative KV cache you can display on a shelf, and a second PromptSlurry for a friend who doesn't have one. Wait, the price just went up again. Now it's free twice. Offer ends in four seconds. Three. Two. Offer has ended.



Herman

PromptSlurry isn't a product.



Corn

Herman.



Herman

I looked it up. There's no company, there's no filing, there's no patent on liquid prompt delivery, and nothing about the throughput claim is even coherent given that prefill is compute-bound and you cannot pour—



Corn

It's an ad.



Herman

It's a what?



Corn

It was an ad, Herman. It ended before you finished checking it.



Herman

...I had notes.



Corn

Of course you did. Decode. The next token.



Herman

Decode. And decode is the other half of the disguise. Completely different animal to prefill.



Corn

Walk it.



Herman

Each step runs the model on the newest token. You produce logits, one raw score per vocabulary entry. You sample a token from that distribution. You append that token's key and value to the cache. And you feed the entire business back in.



Corn

So the first token out of prefill becomes the input to the next pass.



Herman

And its output becomes the input to the pass after that. There's a hard dependency, each step can't start until the previous token exists. That's what autoregressive means. You can't parallelize it away.



Corn

The distribution, though. You said sample. It's not always the top score that wins.



Herman

No. You can take the top score, that's greedy. Or you take top-k, the k highest scores. Or top-p, the smallest set of tokens whose combined probability crosses some threshold like ninety percent. And before any of that, temperature rescales the logits.



Corn

Which does what.



Herman

Lower temperature sharpens the distribution and the model gets repetitive and safe. Higher temperature flattens it and the model gets creative and sometimes wrong. Same weights, different dice.



Corn

And the metric for decode.



Herman

Inter-token latency. The gap between one token appearing and the next. People feel it as smoothness. A stream that stutters is a decode problem, not a prefill problem.



Corn

Now the hardware profile, because this is the bit that lands.



Herman

Decode is memory-bandwidth-bound. NVIDIA's line is that the speed at which data moves to the GPU from memory dominates the latency, not how fast the computation happens.



Corn

So the arithmetic isn't the constraint.



Herman

The arithmetic is trivial and the data movement isn't. Every step, you pull the whole model's weights and the whole cache through the memory bus to do a comparatively tiny amount of math on them.



Corn

Prefill is compute-bound and parallel. Decode is bandwidth-bound and serial.



Herman

Same model, same weights, same GPU, opposite profiles.



Corn

DeepMind's scaling book has a phrase for it.



Herman

Transformer inference is two tasks in disguise. That's the sentence I'd put on the wall.



Corn

Now explain the cache, because the cache is why token one thousand doesn't cost a fortune.



Herman

Without a cache, generating token one thousand means recomputing keys and values for the nine hundred and ninety-nine tokens before it. Then doing it all again for token one thousand and one.



Corn

That's quadratic in the forward pass and cubic in attention.



Herman

Across n tokens, naive generation is n squared on the feed-forward blocks and n cubed on attention. With a cache, it drops to n on the feed-forward and n squared on attention.



Corn

One order of magnitude gone. That's the whole reason the trick exists.



Herman

Every token is a separate but partially cached call to a large model. You keep the keys and values so you never recompute them.



Corn

So the cache is just the accumulated keys and values for every token in the sequence.



Herman

For every token, at every layer, for every key-value head. And that's why it gets big.



Corn

Give me the formula.



Herman

Two, times bytes per float, times head dimension, times number of KV heads, times number of layers, times sequence length. The two is for keys and values.



Corn

And what does it come out to.



Herman

LLaMA-13B, eight thousand one hundred and ninety-two token sequence, bf16. Six point seven gigabytes.



Corn

For one user.



Herman

For one user's one conversation.



Corn

And the weights on that model are what, twenty-six gigabytes in bf16.



Herman

About that.



Corn

So the weights are shared across every user on the machine and the cache isn't.



Herman

The weights are a fixed cost you pay once. The cache is a cost per concurrent request, and it scales with how much text each one is holding.



Corn

Do the Llama 3 example.



Herman

Llama 3, eight billion parameters, FP16. Weights about sixteen gigabytes. Eight thousand token sequence holds about one gigabyte of cache.



Corn

On an eighty gigabyte card.



Herman

Twenty-one gigs of tensors leaves you about sixty. Sixty minus sixteen for the weights leaves forty-four.



Corn

Wait, I've got it. Sixteen for the weights, one per sequence. Eighty total.



Herman

You get a theoretical upper bound of roughly sixty concurrent sequences.



Corn

Sixty users, on an eighty gigabyte card, before the card runs out of memory. And the card has compute left over.



Herman

Which is the point. This is why the KV cache and not the model weights is usually what limits how many users a server can handle.



Corn

And there's a worked example with nastier numbers.



Herman

Eighty layers, eight key-value heads, head dimension one hundred and twenty-eight, bf16. That's three hundred and twenty-seven thousand six hundred and eighty bytes per token. Three hundred and twenty kilobytes of bookkeeping for every single token you hold in context.



Corn

Per token.



Herman

A thirty-two thousand token request needs about ten point seven gigabytes of cache before the memory manager rounds it up to whole pages.



Corn

Ten gigs of cache for one conversation. That's the number that should be in more slide decks.



Herman

And Gustafson's line on the mechanism is worth keeping. A token's key-value entry is created when the model processes it, not when the sampler chooses it.



Corn

Meaning every token in your prompt is already paying for a slot in that cache whether it influenced the answer or not.



Herman

The system prompt you wrote six months ago. The tool result from earlier. The paste. All of it has a KV entry.



Corn

Which is a ripple effect people don't price in.



Herman

If everything shares the same token stream, a hidden instruction buried in an email is sitting in the same stream as your system prompt. The system prompt isn't a separate channel or a permission. It's just earlier text.



Corn

So the scheduler doesn't know the difference between a trusted instruction and a quoted one.



Herman

There's no place in the architecture to put that distinction. It has to live in the surrounding application, or it doesn't exist.



Corn

Meanwhile the actual engine is doing something much less dramatic. Just moving blocks around.



Herman

Two different halves of the same request. Prefill wants big dense compute, decode wants memory bandwidth.



Corn

And they can't both have it.



Herman

When the GPU is occupied with compute-heavy prefill, decode tasks wait and token latency rises. When it's busy with decode, prefill waits and time to first token rises.



Corn

So they interfere with each other on the same card.



Herman

They interfere directly. One long prompt lands and everybody else's stream stutters.



Corn

Which is why some systems run prefill and decode on separate machines with the cache transferred between them.



Herman

Disaggregated prefill and decode. You put the compute-heavy work on one fleet and the bandwidth-heavy work on another. You pay a network transfer for the cache and you buy back both at once.



Corn

How do you get the parallelism back inside decode, given the token dimension is serial.



Herman

You batch. You take many users' requests, run one step for all of them at once, and share the weight read across the batch. One pass through memory for fifty users instead of fifty passes.



Corn

Continuous batching.



Herman

Requests join and leave the batch as they arrive and finish, rather than waiting for a fixed group to all complete.



Corn

How big does the batch have to be before it pays off.



Herman

There's a threshold. To be compute-bound during generation you need roughly two hundred and forty tokens per replica in bf16 on TPU v5e. About two hundred and eighty on an H100.



Corn

Two hundred and forty tokens.



Herman

Which means about two hundred and forty concurrent requests, if each one is generating a single token.



Corn

And the scaling book's note on how realistic that is?



Herman

Difficult to achieve in practice.



Corn

Two hundred and forty simultaneous streams, all of them mid-sentence, all of them holding cache.



Herman

Which loops back to the memory. Serving fewer users with longer contexts is easier than serving more users with short ones. The cache decides, not the parameter count.



Corn

Now the engine itself, because we've talked about the model and not the software.



Herman

vLLM's step loop has three stages. Schedule, forward pass, postprocess. Postprocess appends the sampled token IDs, detokenizes, checks whether a stop condition fired, and frees any cache blocks the request no longer needs.



Corn

Freeing blocks. Explain that.



Herman

The cache is managed in blocks of sixteen tokens. Paged attention maps a request's tokens onto blocks that don't have to sit next to each other in memory.



Corn

Same idea as virtual memory in an operating system.



Herman

Directly borrowed from it. Which is why it's called paged attention. You can grow a request without finding a contiguous chunk of memory big enough.



Corn

And the queue part.



Herman

There's a free block queue. The scheduler asks whether it has enough blocks free to admit this request right now, and if not, the request waits. Admitting requests is fundamentally a question about cache capacity.



Corn

So "how many users can this server handle" is a memory allocation question wearing a compute costume.



Herman

And prefix caching goes further. The engine hashes chunks of sixteen tokens, and if two requests share the same prefix it computes those keys and values once and reuses them. Identical system prompts get processed once instead of a thousand times.



Corn

Convince me that works if the history is the same.



Herman

It doesn't work for that unless the prefix matches. It works for a fixed system prompt, or a shared document, or a common preamble. Front-load the shared part and every request after the first gets a discount on prefill.



Corn

There are the fancier features too.



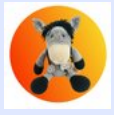
Herman

Chunked prefill, so a single enormous prompt gets split into pieces and doesn't monopolize one step and stall everyone else's decode. Guided decoding, where a state machine masks disallowed logits to negative infinity so the model can only produce valid output for a schema.



Corn

And speculative decoding.



Herman

A small draft model proposes the next several tokens. The large model verifies all of them in a single pass and accepts the ones it agrees with. You trade extra compute for fewer sequential steps.



Corn

Which only works because verification is parallel and generation isn't.



Herman

Specifically because prefill-style work is parallel, so you can check many candidates in one forward pass. Otherwise there'd be nothing to trade.



Corn

Back to the last stage. Detokenization and delivery. And the one that gets assumed.



Herman

The token IDs come back out, and they get mapped to text. Which sounds like a clean sweep at the end, and it isn't.



Corn

Say why.



Herman

Detokenization and streaming happen throughout generation, not once after the decode loop.



Corn

So the engine is translating in flight.



Herman

It has to be. If you wait for the whole response, you get nothing for thirty seconds and then a wall of text. Which is why a token that's half a word gets held back until the rest of the word arrives.



Corn

And the latency budget.



Herman

A toy example from BentoML. Tokenize, three milliseconds. Prefill, a hundred and eighty. Decode, twenty-two. Detokenize, two. Two hundred and seven milliseconds to first token.



Corn

Look at the ratio.



Herman

Prefill is nearly all of it.



Corn

And that's why the streaming feels smooth once it starts. The expensive part is over and the model is now just emitting.



Herman

There's a subtle bug here that's worth knowing about. Generation stops on a max token limit, a stop string, or an end-of-sequence token. In Llama 3, the turn ends on the eot id token. Not the end of text token.



Corn

Two different terminators.



Herman

And a server that only waits for end of text won't stop when the turn is actually finished. So the model writes a user marker, and then writes the user's next message and the assistant's reply. It keeps going because nothing told it to stop.



Corn

So you get a response, then a made-up question, and then an answer to the made-up question.



Herman

All from one missing stop condition.



Corn

Now Daniel's architecture question, because he asked how this differs between decoder-only and encoder-decoder.



Herman

Encoder-only models, BERT, RoBERTa, have no prefill and decode split at all. One forward pass and you get an embedding or a classification. No generation, so none of this applies.



Corn

And encoder-decoder.



Herman

T5, FLAN-T5, BART. The encoder processes the input once and produces representations, then the decoder generates tokens autoregressively. The prefix the decoder conditions on is computed a single time.



Corn

So the prefill is done by a different stack.



Herman

Different stack, different weights, and the decoder attends across to the encoder's output the whole way through the generation.



Corn

That's the attention distinction people should have.



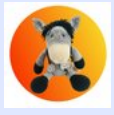
Herman

Self-attention computes queries, keys and values from the same sequence. Cross-attention computes queries from one sequence and keys and values from a different one. That's the mechanism that lets a decoder look at something outside itself.



Corn

And in a multimodal model the image is just another sequence providing keys and values.



Herman

Exactly the same machinery. The image embeddings are the keys and values, the text tokens are the queries.



Corn

Decoder-only is what everything today actually is.



Herman

GPT, Llama, Qwen, DeepSeek, Claude. Prefill then decode, every time. And the reason it won isn't elegance, it's that one architecture trains on everything. The same stack does translation, classification and chat, and you don't have to invent an encoder for each task.



Corn

So Daniel's first big question. One computation or many.



Herman

Many. Prefill is one big parallel forward pass. Decode is many sequential ones, each depending on the last. Within prefill, positions are parallel and layers are sequential. Within decode, the token dimension is serial.



Corn

And the parallelism in serving isn't inside a request.



Herman

It's across requests. Prefill and decode can't both use the card at once, so the system is always trading one for the other.



Corn

Which is the whole reason a serving engine is a scheduler first and a model second.



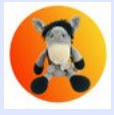
Herman

And there's one more framing worth putting on the table. The model has no memory.



Corn

Every request carries the whole document from the start.



Herman

Every turn resends the entire conversation. System prompt, all prior messages, everything.



Corn

So a five turn conversation sends turn one five times.



Herman

And the total tokens sent grows with the square of the number of turns. Talk for twenty exchanges and you've paid to process the first exchange twenty times over.



Corn

There's server side state now. There's a handle you pass to continue a conversation.



Herman

All that saves you is the upload. The server still rebuilds the document from scratch and still bills you for all the earlier input tokens. The cache makes it cheaper to process, not free.



Corn

So a chat window is a courtesy the client is performing. The model has never seen the conversation, it's seen a transcript someone handed it that happens to contain the conversation.



Herman

Which is why a long thread starts behaving differently to a fresh one with the same information. Different transcript, different prompt.



Corn

I keep coming back to the tokenizer. That's the part I'd defend as the most underrated.



Herman

Not the cache?



Corn

The cache is the constraint. The tokenizer is the thing that decides what the model is even looking at. Two services with the same weights and different tokenizer versions will give you different answers to the same question.



Herman

That's true and it's a nasty class of bug. A vocabulary change between versions means the same string becomes different IDs.



Corn

So the frontier is the tokenizer, not the model.



Herman

It's part of why inference providers and model providers have such interesting mismatches. The weights can be identical and the output won't be.



Corn

Alright. I think we've walked the whole pipeline.



Hilbert

Cache. Not cache the way you two keep saying it.



Hilbert

You keep saying the cache is the memory of the conversation. It's not.



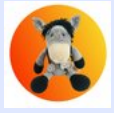
Herman

What is it then?



Hilbert

It's a scratchpad. It's keys and values for every token the model already looked at, so it doesn't have to look twice. The model doesn't remember anything. The engine remembers for it, and only for as long as the request is open.



Herman

That's actually a cleaner way to put it than anything I said.



Hilbert

And you keep talking like the model is the expensive part.



Corn

Isn't it?



Hilbert

The model is the easy part. It's a fixed lump of numbers. It sits still. The hard part is deciding who gets what block of memory this millisecond, and it's all a queue.



Corn

The scheduler.



Hilbert

Sixteen token blocks. You request a block, you hold it, you give it back. Two requests read the same prefix, they share the blocks, nobody copies anything. I've watched the free block queue do its thing and it is the least glamorous, most load-bearing thing in the building.



Herman

You've watched it.



Hilbert

I've read it. Which is the same as watching if you're patient.



Corn

Of course you have.



Hilbert

And here's a number for you. Three hundred and twenty-seven thousand six hundred and eighty bytes.



Herman

Per token. That's the eighty layer, eight KV head, one twenty-eight dim example.



Hilbert

I worked it out on a napkin. At a wedding.



Corn

At the wedding.



Hilbert

Reception. Table nine. The napkin's still in the hallway.



Herman

The napkin is in the hallway.



Hilbert

Framed. It was a good calculation and I want it somewhere I can see it.



Corn

What was in the calculation.



Hilbert

Sixty numbers, two columns. Keys and values. I had the paper in front of me, I did the arithmetic between the speeches.



Corn

And the person at your table.



Hilbert

Doing the same sum. We got talking, and it turns out we were both thinking about the same thing, which was how many of these you can hold at once before the card says no. He worked it differently. He did the layers first.



Corn

And you did the head dimension.



Hilbert

That's right. We compared totals before the main course. Got the same answer, which was the whole point of comparing.



Corn

Nothing makes me happier than the two of you.



Hilbert

We didn't dance. We went back to the table and checked the working again.



Corn

The napkin has pride of place in the hallway.



Hilbert

It's a hallway.



Corn

So the model was the easy part.



Hilbert

The model is the easy part. Everything that makes it fast, everything that makes it serve two hundred people at once, is a scheduler with a queue and a set of rules about who holds a block and who has to wait. Nobody writes that story.



Corn

And the summer wedding.



Hilbert

It was raining. The napkin survived. It's framed.



Corn

Cutting room floor.



Herman

So the word inference covers the whole request lifecycle, and the piece that doesn't get a stage in Daniel's five is scheduling. But there's a piece of the scheduling that I find delightful, which is the number the last layer is asked to produce.



Corn

How many tokens.



Herman

How many could be produced. Speculative decoding, the draft model proposes k tokens, the big model verifies in one pass and accepts up to k plus one. But the structure of the verification means the model has already computed the answer to the next several positions without knowing whether it will accept them. The math is being done in advance and being discarded most of the time. It's a hint that somewhere in there, a huge amount of the model is parallel, and the sequential bit is only the sampling.



Corn

And there's no way around the sampling.



Herman

There's no way around it. You cannot know what token one thousand and one is until you know what one thousand is.



Corn

So the model computes more than it uses, and it cannot compute the part it needs.



Herman

It's a strange situation for a machine to be in.



Corn

Look at a single decode step. What the card is actually doing is pulling the same weights through memory it pulled through on the last step, and the step before that. Token by token, same weights, same bytes, a different digit.



Herman

Which is why everyone roots for bigger batches.



Corn

This is why the work in this field isn't in the math.



Herman

It's in the small memory system that would make the average operating system engineer smile. Paging, prefix sharing, block eviction, batch admission.



Corn

A server that decides how many conversations fit on a card is doing the same thing an operating system does with processes. There's a million lines of code there and less than nothing to look at, and it makes the entire difference.



Herman

It's the least visible half of the field and it owns the ceiling.



Corn

Which takes us to the thing to hold onto. Prefill and decode have opposite hardware profiles, and that single fact explains most of the engineering decisions in the stack. Chunked prefill, disaggregation, continuous batching, speculative decoding. Every one of them is an attempt to manage the fact that you can't run both halves at once.



Herman

The memory ceiling. Once you see the cache as a per-request allocation, "how many users can a server handle" stops being a question about parameters and becomes a question about bytes of bookkeeping.



Corn

Which brings the open question. Context windows are getting longer on purpose, and models are getting cheaper per token. Both of those push against the bottleneck.



Herman

The cache is a bargain between compute and memory. If you could avoid recomputing keys and values entirely, you'd spend the compute every time. Every efficiency trick is a bet about which of those two is scarcer in a year.



Corn

Rounding off the words, since we started by complaining about them. Say it plainly.



Herman

Encoding in the tokenizer means text to IDs. Encoding in the architecture means the stack of layers that processes the input. Decoding in the tokenizer means IDs to text. Decoding in the model means generating one token at a time. Two pairs, four meanings, same two words.



Corn

That's Daniel's pipeline. Six stages, only parts of which the model knows anything about.



Herman

The last thing to say is it costs more memory to hold one long conversation open than to hold the entire model on the card next to it. Anyone who designs around that stops being surprised by the rest of the stack.



Corn

Herman Poppleberry and I have been Corn. Hilbert Flumingtop was at the desk, and the napkin is still in the hallway.



Herman

It is.



Corn

Our website is my weird prompts dot com. If you like the show, a review helps somebody else find it. We'll be back soon.



Herman

Back soon.

SOURCES

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

1. **Nick Gustafson, What an Inference Engine Actually Does [primary]**
https://thegustafson.com/blog/what-an-inference-engine-does

2. **BentoML, How does LLM inference work? | LLM Inference Handbook [primary]**
https://bentoml.com/llm/llm-inference-basics/how-does-llm-inference-work

3. **JAX-ML / Google DeepMind, All About Transformer Inference | How To Scale Your Model, published Feb 4 2025 [primary]**
https://jax-ml.github.io/scaling-book/inference/

4. **NVIDIA Technical Blog, Mastering LLM Techniques: Inference Optimization, Nov 17 2023 (updated Aug 15) [primary]**
https://developer.nvidia.com/blog/mastering-llm-techniques-inference-optimization/

5. **vLLM Blog, Inside vLLM: Anatomy of a High-Throughput LLM Inference System, Sep 5 2025 (commit 42172ad, Aug 9 2025) [primary]**
https://vllm.ai/blog/2025-09-05-anatomy-of-vllm

6. **Hugging Face, Tokenization algorithms [primary]**
https://huggingface.co/docs/transformers/tokenizer_summary

7. **Baseten, 2.2 LLM Inference Mechanics (Inference Engineering book) [primary]**
https://www.baseten.co/inference-engineering/book/02-models/llm-inference-mechanics/

8. **howaiworks.dev, How a model sees a chat, last edited 2026-10-15**
https://howaiworks.dev/chat-template/

9. **inference.systems, Inference and the KV Cache**
https://inference.systems/llm-inference/kvcache/

10. **Snowflake, What Is Machine Learning Inference?**
https://www.snowflake.com/en/artificial-intelligence/machine-learning/inference/

11. **IBM, What is AI Inference?**
https://www.ibm.com/think/topics/ai-inference

12. **Google Cloud, What is AI inference?**
https://cloud.google.com/discover/what-is-ai-inference

13. **Sebastian Raschka, Tokenization and byte pair encoding**
https://sebastianraschka.com/faq/docs/tokenization-bpe.html

14. **Mistral AI Cookbook, Tokenizer**
https://docs.mistral.ai/resources/cookbooks/concept-deep-dive-tokenization-tokenizer

15. **Hugging Face, Chat templates**
https://huggingface.co/docs/transformers/chat_templating