Computation and Data Movement for Inference

Mapping MoE models onto inference hardware: structure, flow, and efficient serving

TANJ 3E\NETT

SEP 22, 2026 · PAID

153 2 Share

Mixture of Experts, now widely used in frontier models, has changed both the structure of serving and the economics of useful inference. It did more than increase parameter count. It changed which tensors are active for each token, what must remain close together, which transfers need strong local bandwidth, which can tolerate a weaker network link, and how memory movement, storage, and scheduling contribute to useful throughput.

The best place to begin is the service as a whole. Inference runs inside a cluster coordinated by an orchestration layer such as NVIDIA Dynamo, Mooncake, or a custom scheduler. These work closely with inference servers like vLLM or SQlang, and those also have their own orchestration features. This article will not go into the details of how you work with the orchestration software or which you should choose. The article aims to provide an overview of the process and reasons for various features you may use.

A user (or their agent) starts a conversation with a request (query), and the conversation may continue after answers with more requests. The request- answer is a “turn”. In modern AI systems the user is often running a in a client application (whether GUI or command-line oriented) and some of the answers coming back from the AI are intercepted by that application as instructions for it to run, things like editing your code or searching your company guidelines on an HR question.

The AI system can also do some of those things itself from its data center, for example it may search the web. The results from these intercept tool actions are also returned to the AI as requests, creating more turns. The server keeps a context (often referred to as KV cache or just cache) which is the distillation of the session, allowing each new request, from user or from tool, to be properly interpreted for overall forward progress. There can be thousands of turns per hour when the user launches an agent on a long-running task. The conversation can also pause and resume hours or even days later.

When a request arrives at the inference servers the orchestration puts it into a queue, while will feed into an input-fill worker, along with any context from system prompts and prior turns in the conversation. The query is then converted into new context appended to the conversation. That new context state is then moved to a decode worker. Decode repeatedly reads the accumulated state, generates answer tokens, and those answer tokens are also appended to the conversation state. The process may loop through tools, agents, users, and further input-fill work. A model worker running on GPUs is only one part of this token factory; storage, networking, and orchestration connect its stages.

Inside the workers, four operating regimes are worth distinguishing from the beginning:

1. **Prefill**, where the initial block of new tokens is processed together.
2. **Midfill**, where a continuation request is appended to an existing cached context.
3. **Decode attention**, where the context is used to generate the basics of newly generated tokens.
4. **Decode experts**, where each basic new token is refined by a selection of experts a large total set of possible experts.

These regimes place different demands on compute, memory, and networking. Prefill and midfill are closely related, with Prefill being a Midfill with zero prior context. However, Prefill is a common special case, corresponding to one-shot queries like classic “”chatbot” use, and can be optimized a bit differently than Midfill. Prefill reaches high arithmetic intensity (ratio of computation to data movement) and does not need to wait for context to be located and read. Midfill begins from an existing KV cache for prior state and appends a new request input sequence. This generally has a more moderate arithmetic intensity than prefill because there are fewer new tokens and more existing data to move per token.

Decode attention and decode expert work are generally low arithmetic intensity since there are few new tokens compared to the prior context or expert weights, which at decode state are data that needs to be moved. Meanwhile, across all of them, the tensors for attention and for experts are the same regardless of token value, so there is benefit to sharing a single reading of a tensor with as many tokens as can be queued to use it, even tokens coming from unrelated requests by other users which simply happen to be running at the same time and which are networked closely enough to

share. Treating all four mode as the same workload gives away much of the structural advantages that MoE models offer in terms of different intensity and different sharing patterns.

In this document we will treat the 4 stages separately, and sometimes assume they are disaggregated. There are tradeoffs between aggregated (where the models stay on one server, which reconfigures as the request progresses between the stages) and disaggregated where the orchestrator may find a free slot on another machine already configured appropriately. Generally we will talk about the stages as disaggregated, with aggregation being a special case of always making the current node available when a previous stage finishes. There will be a section on the trade-offs at the end, after the work to be done has been more completely introduced.

A transformer model is a deep stack of repeating layer groups. A group may contain one kind of layer, or it may contain a full-attention layer together with several linear, local, selective, or otherwise optimized-attention layers (this is a hybrid group of layers). Within each layer, attention, shared transformations, routing, selected experts, and recombination occur in sequence. Those steps are known in advance and repeat continuously, allowing the same hardware resources to serve different parts of the flow at different moments.

The machines can also be described equally simple recurring units. A node is a group of accelerators, typically a tray in a rack, each accelerator pairs compute with local fast memory. A node is coordinated by a CPU and connected to other nodes through NICs (network interface controller). Nodes stack into racks. A rack may act as one scale-up domain when very large tensors are , several smaller scale-up islands, or a set of pipeline stages. Datacenter networking then joins bounded workers to storage and orchestration rather than participating in every inner tensor operation.

This essay follows that progression. It begins with the global service and the operating regimes, then opens the model into wide layer-group slices and maps them onto accelerator nodes and racks. From there it develops pipeline, tensor, and expert parallelism; the limit imposed by KV-cache memory; the different value of batching in prefill, midfill, and decode; and the role of scheduling in a many-rack token factory.

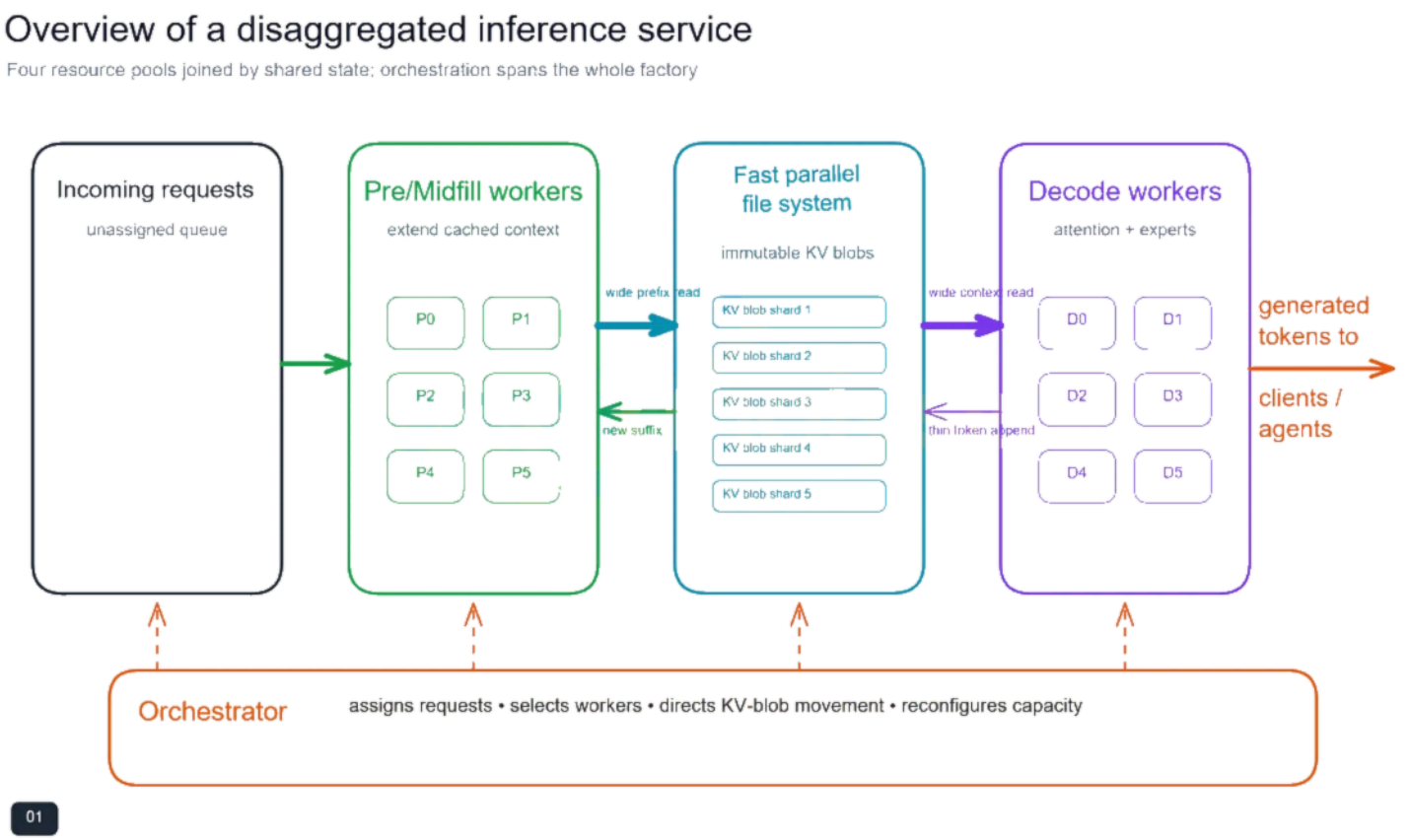


Figure 1. Overview of model stages in an inference service.

1. Orchestration delivers requests via queues into pools of workers

An inference service is a cluster where prior requests and immutable state are stored when idle and then queued for delivery to specialized workers when a new request pulls in context or a prior request reawakens for continuation. The incoming queue is not yet a batch. Each request carries prompt tokens, references to reusable context, service objectives, and often an existing conversation or agent state. An orchestrator may place it into a prefill, midfill, or decode batch, and the request can join or leave that batch independently as it completes, making that capacity available for new work. Continuous batching works best when the scheduler knows which workers have slots that match the request's context length and service objective.

Prefill workers create context from a substantial block of new tokens. Their output is new KV state for every model layer, stored independently of the worker that created it. The KV state moves along model layers, the output of layer K from one turn becomes input to layer K in the next turn. In pipelined servers this creates a natural "torrenting" of output and input from separate NICs.

Midfill workers extend previously processed context. The cached prefix may contain system and user prompts, memories, earlier conversation turns, agent steps, or

retrieved documents. Large roots may be already cached, while the incremental input can be much smaller than a comparable uncached request: a half-million context may receive only a few hundred or a few thousand new tokens.

Decode workers consume the processed, accumulated context and generate one or a few tokens (multi-token prediction is becoming quite common) per pass. Every generated token adds a small amount of new state. The same context can later return to a midfill worker when a user, tool, or agent contributes another block of input. Agentic work using tools creates a repeated sequence of context extension and generation, not merely one prefill followed by one decode.

A worker may occupy a node, a tray, or a rack. The service becomes large by operating many bounded workers and moving requests and state among them. Depending on precision and working-state requirements, even multi-trillion-parameter models can fit within the aggregate memory of a modern rack-scale system. The datacenter network remains essential because the factory must connect many such workers to shared storage and continuously match worker configurations to demand. KV transfers may be gigabytes in size, but they can be striped or torrented across multiple links and destinations; their movement time can remain well below the lifetime of a long decode job.

This separation is useful for both performance and operations. Prefill and midfill are predictable once the cached-prefix size, new-token count, and worker configuration are known. Decode completion time is stochastic because arrival of the final output token cannot be predicted in advance. Shared memory and storage between the pools let each stage run at its own cadence. Later sections of this article will discuss queues, ready buffers, and worker reconfiguration. For now, the important point is that the service is a loop of stages joined by movable state.

AgentX examples

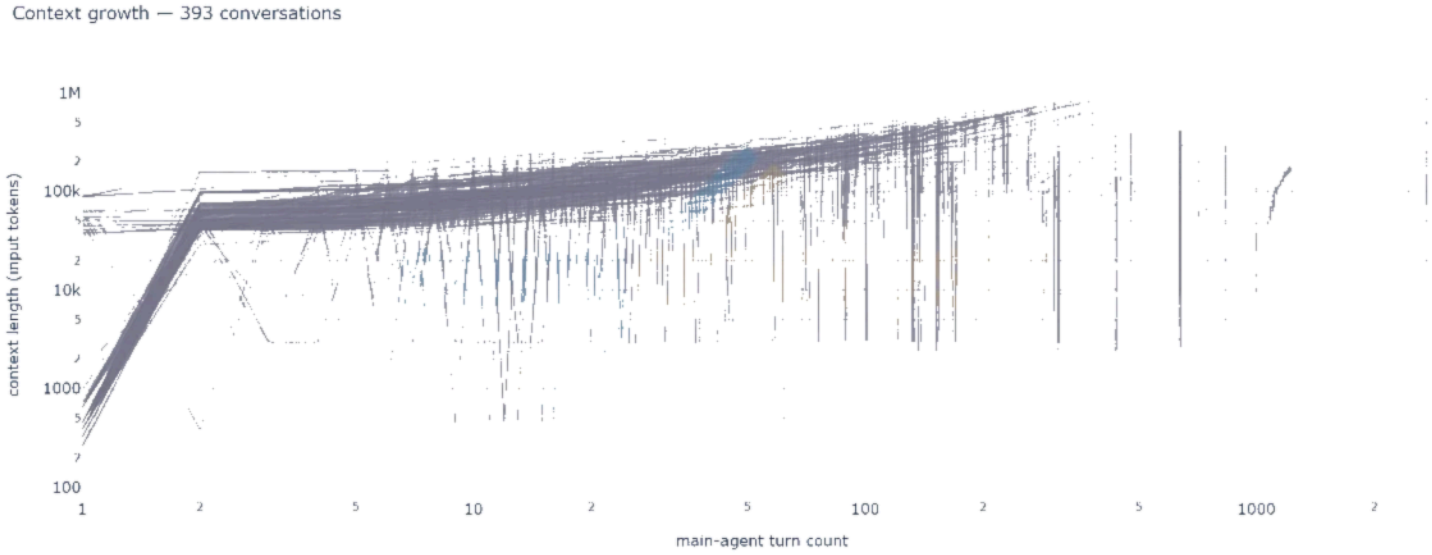


Figure 2. "CC Traces Weka" dataset contributed to AgentX, release 34. Source: SemiAnalysis AgentX.

This is viewed with the SemiAnalysis Conversation Explorer. It shows that context growth during conversations can be quite rapid but also goes through large fluctuations due to compaction and other model behavior which may not be explained in public discussions. Orchestration of the inference context is a competitive advantage for AI companies.

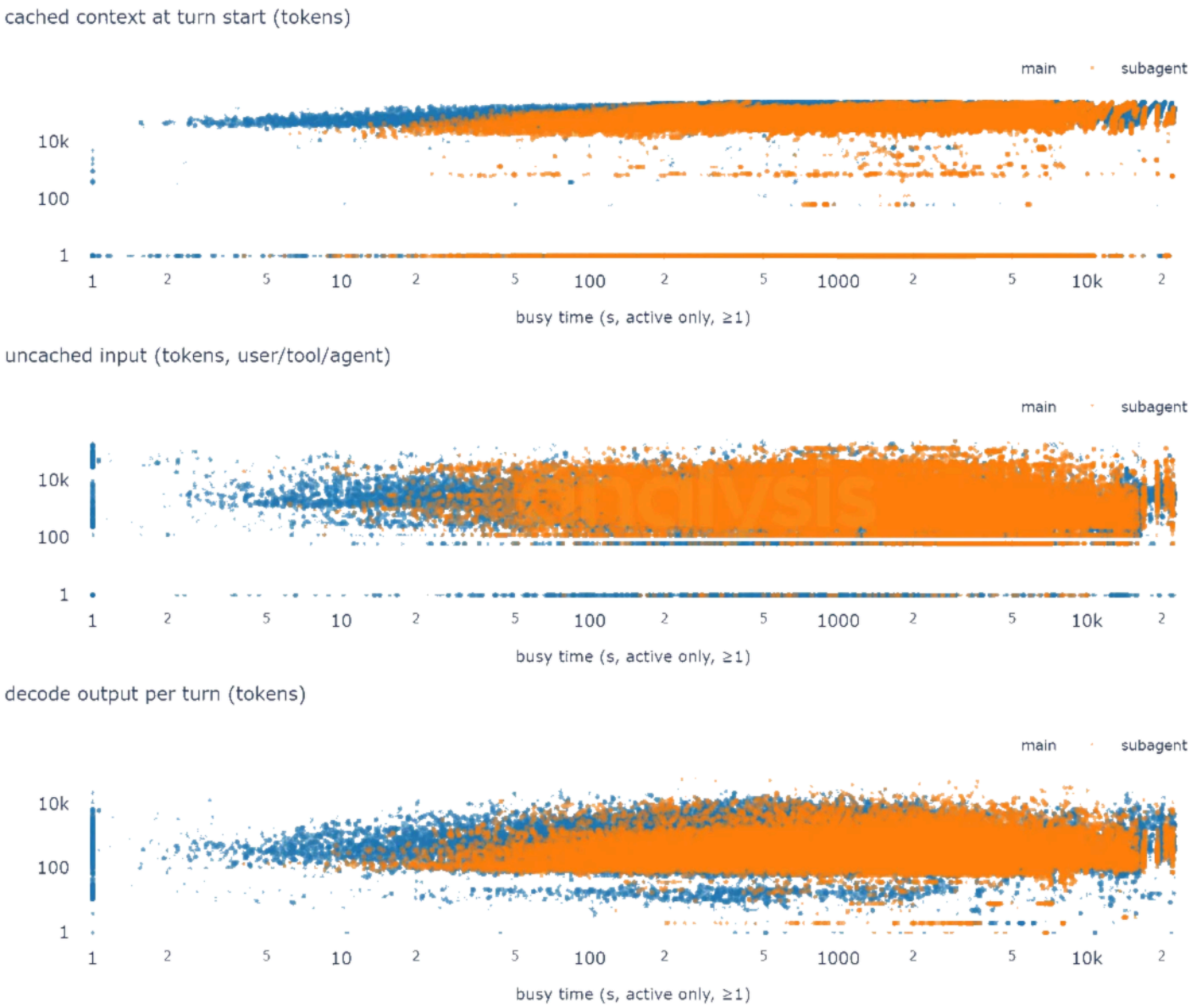


Figure 3. Cache, new input, and result lengths seen at each turn of the same conversations. Source: SemiAnalysis AgentX.

We can take that same dataset and pull out the cache, new input, and result length values seen at every turn. This visual shows how the Token Factory needs to serve many different workloads. In principle there could be requests similar to each dot in those traces (and more dots from other workload datasets) all running at the same time, assigned to some worker in the AI cluster.

This article aims to survey some of the major functions in inference that make it possible,

2. KV state moves as immutable blobs

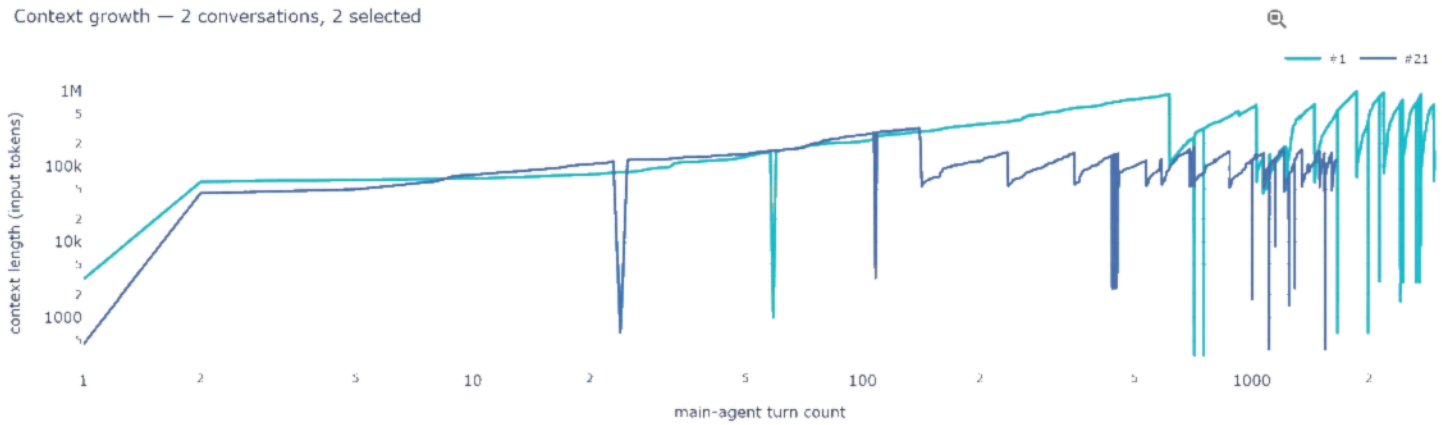
Reusable context at data-center scale becomes a shared object-store of immutable blobs rather than a file attached to one machine. A system prompt, project context, previous user turns, tool output, and generated tokens can be in separate blobs. A new operation reads the blobs it needs and appends new ones; existing blobs normally

remain unchanged. This forward-only structure follows the causal transformer itself. Changes can be handled by backtracking and creating a new branch rather than rewriting the common path.

The durable source of these blobs is a fast, parallel, scale-out memory and storage pool. It needs enough aggregate bandwidth and network reach that prefill, midfill, and decode workers can be selected for suitability and availability rather than because one machine owns the only copy of the context. Newly idle blobs can first move into shared network-attached DRAM, including node-CPU memory and dedicated memory appliances. As that tier fills, a classifier can discard blobs that are not likely for reuse, or promote longer-lived state to SSD. The underlying text and references are often orders of magnitude smaller than the expanded KV representation, so it is possible rebuild a KV blob from that smaller text if the triage decided to reuse space and discarded the expanded embedded state. AI calculations can vary subtly, so while the rebuilt state can be expected to be a valid context, services may take different approaches to how casually they discard and rebuild, and how much they try to keep important immutable state.

For a deeper dive on how the blobs are managed, [Unified Radix Cache: One Tree for Hybrid Model Prefix Caching - LMSYS Org] is recommended reading. It is not the last word, but it is clearly written and will point you to other sources if you want to look for more.

There are also frequent compactions of state with long-running agents that build up to the maximum context, so in general after a compaction there will be many prior blobs which have been abandoned in favor of a new context. The space from the pre-compaction contexts is probably recycled for new work.



	#	model	turns ▼	initial ISL	max ctx	final ctx	avg out	max out	hours
	filter dat								
☑	1	claude-fable-5	2,977	3,200	989,824	160,128	1347.6	21,025	216.15
☑	21	claude-opus-4-8	1,647	448	324,544	122,368	579.3	13,365	152.48

Figure 4. AgentX release 34, selected conversations. Source: SemiAnalysis AgentX.

You can see compactions occur in the right half of these conversations as contexts get close to the practical context limit for each model (250k tokens for Opus 4.8, 1MT for Fable). There are also "stalactites" in both lines which seem transient and reflect some proprietary behavior in Anthropic models.

Accelerator HBM is the "hot" working tier. It is too expensive and supply-constrained to be a good default for passive context storage. The active prefix and suffix should enter HBM shortly before use and leave promptly after the worker has finished with them. CPU DRAM is a useful staging and assembly tier, especially for outgoing state: a completed worker can move newly generated blobs into CPU memory while the storage system chooses placement and redundancy, after which the NIC sends them into the shared pool.

For incoming data, RDMA-capable systems may allow the network to place data directly into accelerator memory, avoiding a full copy through CPU DRAM. CPU memory still remains useful for metadata, coordination, partial assembly, fallback paths, replicas, and outgoing staging. When a worker uses multiple GPUs the incoming context can be striped directly to their destination memories in parallel.

Rack-level hot-blob caches can also be useful, whether implemented as memory/storage appliances or making use CPU memory which is assigned to the distributed storage pool. Very common objects such as system prompts are natural candidates. These caches should remain shared scale-out resources rather than private state that binds a request to one decode machine. HBM is several times more expensive and more supply-constrained than DDR. HBM's best use is for data ina an active batch currently earning revenue.

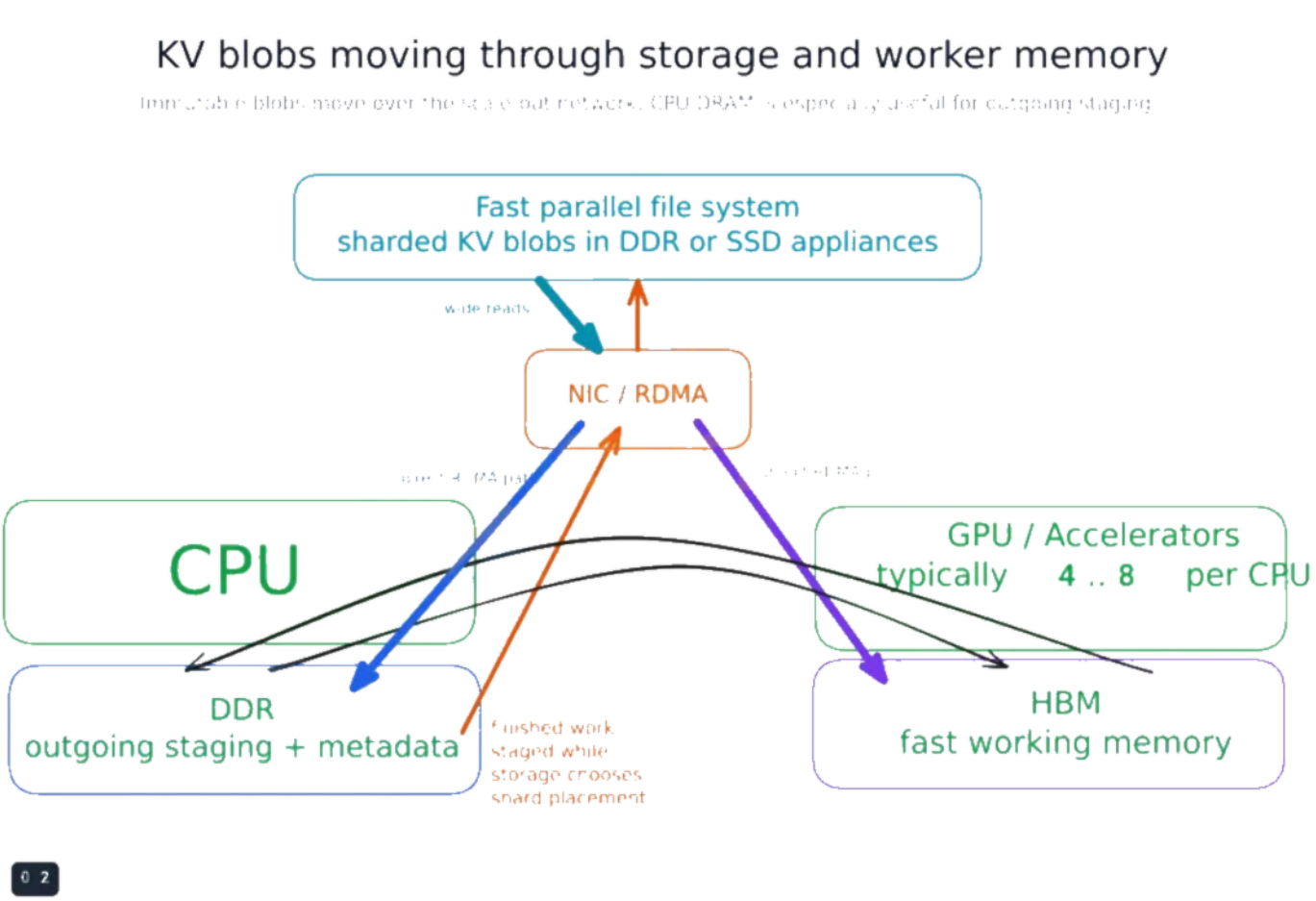


Figure 5. KV blobs moving through storage and worker memory.

Data input is generally larger than output. Midfill commonly reads a large prefix and appends a meaningful but smaller suffix. Decode reads the accumulated context and appends only the state for one or a few generated tokens. The reads are repeated, the writes usually occur just once. The system design should show these differences directly with the width of each data path.

3. Prefill, midfill, and decode create distinct operating regimes

The global service contains several computational regimes. Their differences are more useful than the generic labels *compute-bound* and *memory-bound*, because each has a different opportunity for reuse and a different natural mapping onto hardware.

3.1 Prefill

Classic prefill begins with little or no reusable KV state and processes a substantial block of new tokens. Those tokens share weight loads, use matrix operations efficiently, and create enough routed activations to form useful shared-expert batches. Attention and dense transforms can reach high arithmetic intensity, so prefill is often

constrained primarily by compute. Long initial input contexts can raise memory traffic but these rarely get to 100,000 tokens in length, while midfill is commonly beyond that.

3.2 Midfill

Midfill appends new tokens to a much longer cached prefix. The new block can still provide hundreds or thousands of activations for shared weight and expert reuse, raising arithmetic intensity to thousands of ops per byte of data. Attention must also read a large existing KV state, which can be tens of GB. It therefore sits in a mixed regime: more arithmetic reuse than decode, but much more cached-state traffic than an initial prefill of the same new-token count.

3.3 Decode attention

Decode advances on input of one or a few tokens per query. Each query brings its own KV state which is similar in length to what midfill processes. While prefill differs from midfill on length of cache, decode differs from midfill in input sequence length. These one to few input tokens keep arithmetic intensity low, this work is clearly dominated by the memory movement around the KV cache.

For long contexts, the attention read can exceed the size of tensor operations associated with model weights. Attention is therefore one of the largest operations in modern decode even though most model parameters reside in experts. Delta, top-k, or linear-attention algorithm layers can reduce the burden, but periodic use of a layer with full attention remains a major memory load. Each query brings its own context, so batching queries does not change the arithmetic intensity of attention.

3.4 Decode experts

Experts are not specific to the decode phase. Everywhere the model runs the experts run, and they require no context other than one token's embedding. The history of the query has already been compressed into the activation presented to the expert. This allows various designs that share the experts across all available GPUs, reducing memory size per GPU and increasing the memory bandwidth / memory size, or memory intensity ratio. You can look at the interval needed for a GPU to read all its experts as the best case interactivity (token rate seen by each user) and so if the memory bandwidth remains constant but the GPU is responsible for fewer experts, interactivity can be higher. The gotcha is that networking needs also rise, and the all-to-all pattern needed for routing to experts is a difficult one.

At the end of attention (in prefill, midfill, or decode) a router calculation (a small tensor operating on the unrefined token) selects a few experts for each new token. Tokens that select the same expert can share a tensor-weight load even when they come from different requests, so long as they can be gathered together in time to take advantage of the same weight loading. In practice their may be 15,000 experts across all the layers, so taking advantage of such coincidences requires deliberate synchronization like awaiting until all queries in a batch have completed attention until switching over to sending them for expert work, and that in turn may be improved by having multiple instances in different nearby machines also synchronized to that schedule. If you have ever wondered why Nvidia makes such an effort to connect 72 GPUs so closely, this is a big part of why. In a large token factory it is feasible to have those 72 machines all running in sync, allowing the experts in one layer to be divided up to 72 ways. If a layer of the model has 256 experts then each GPU is handling only 3 or 4 in each layer, permitting very fast complete cycling through all the experts. And a lot of all-to-all traffic.

Something interesting happens if your experts can all be kept in SRAM. You need a huge number of accelerators and a monster network to collect them all to all back to the GPUs, but now there is little reason to wait for a batch to form. Each expert can run whenever it has input. The energy per byte loaded is as much as 100x better than loading from HBM, so a single token running can be as efficient as perhaps a queue of 100 was before. In return for this DAF (disaggregated attention-FFN, where the experts are the FFN) you can free up the instances from needing to be synchronized. This advantage does not help as much with prefill or midfill since they trivially have a batch of tokens they can sort into shared queues on one layer, but it still may offload all the expert weights from those GPUs allowing them to optimize their local memory for other uses like longer contexts. The networking is still a gotcha, where an SRAM based accelerator could turn around an expert calculation in less than a microsecond but now you may be flooding a switch connected to hundreds of nodes with the all to all traffic for millions of tokens. The network may absorb more money and power than the experts they connect.

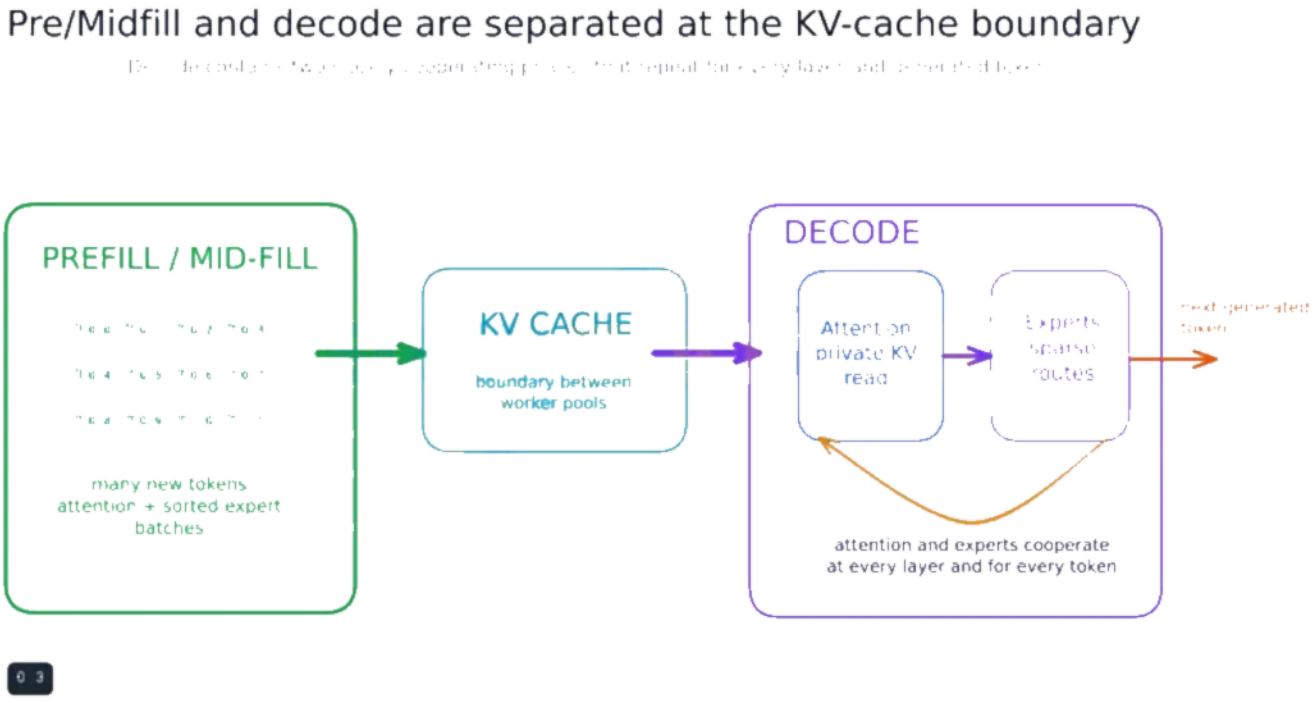


Figure 6. Prefill and midfill are separated from the closely coupled decode loop at the KV-cache boundary.

Inference workers can improve sharing by coordinating many attention instances on the same model layer. Their routed activations then draw from one layer's expert bank, reducing the number of different experts each destination must load. In a pipelined server this also means the networking connections used for expert parallel can be local to that stage of the machine, there will never be a routing that needs experts from another layer. Even so, each decode request contributes only one new token with typically 8 to 16 routed experts per pass, and a layer may contain hundreds of experts. At small batches, sharing remains limited and memory bandwidth continues to dominate both the KV scan and the active expert loads, with very little computation per byte of weight loaded. Expert memory needs to be very low energy per bit, no matter what kind of memory is in use.

These regimes explain why prefill, midfill, and decode may prefer different worker configurations; why decode attention and decode experts can prefer different placements within a node; and why batching helps each stage by a different amount. As prefill in modern agent work rapidly expands to the 500k token range there is a lot of memory movement, especially in token decode - the stage where the model is generating useful intermediate and final results. Fast and large 3D RAM will be decisive in efficiency. Fast memory is also useful for midfill. However, capacity needs to reach a minimum level for the large contexts we now see as normal. If memory capacity is too small then the losses in networking and excessive swarms of chips can waste power compared to future memory types which will use 3D capacity expansion without giving up on throughput.

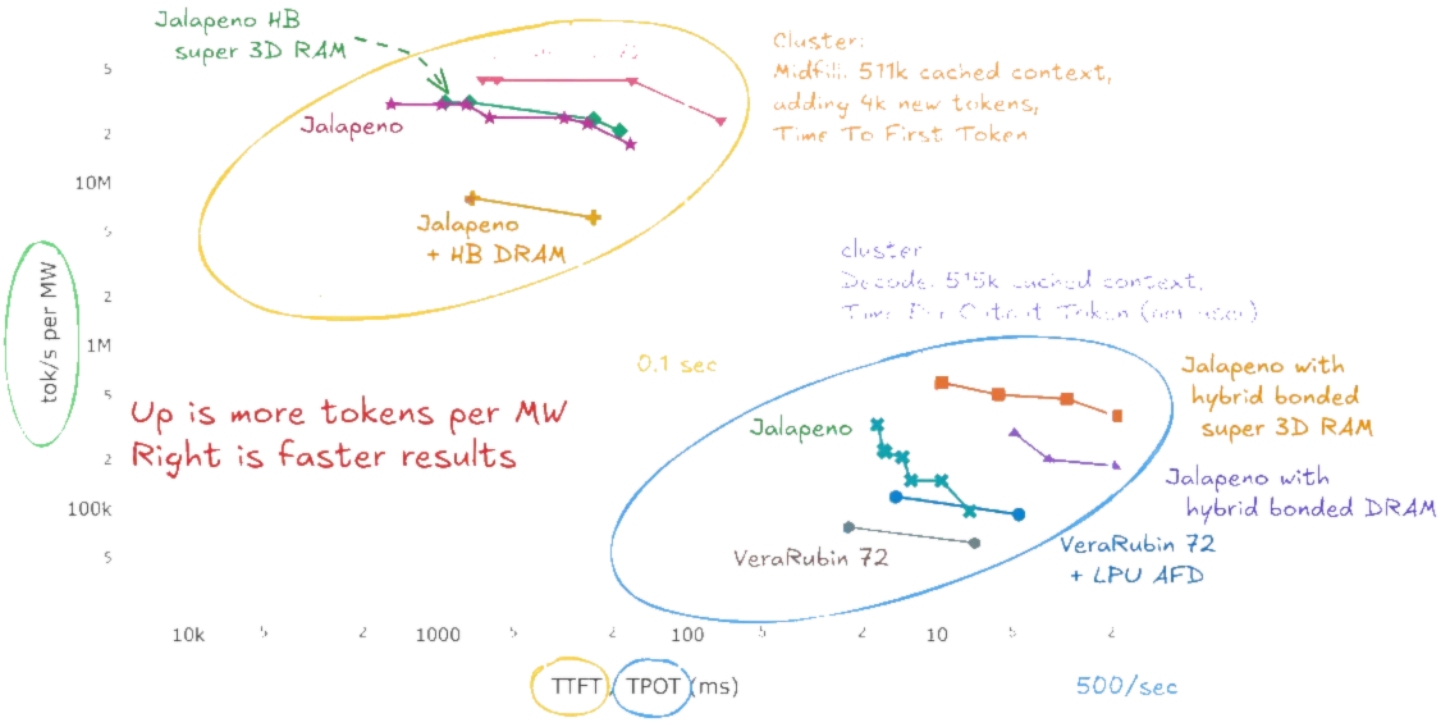


Figure 7. Projected effect of faster memory (hybrid-bonded DRAM, super-fast 3D RAM)) on midfill and decode with 500k token context cache length. Source: SemiAnalysis model simulator with projected accelerator and GPU performance.

While Vera Rubin 72 can crush at the mid-fill layer, which is arithmetic intense, it is not so effective at the switch to memory intensive decode where the Rubin cannot fully exercise its computational prowess. With the large context there is a premium on keeping the decode work in the same node and the midfill, avoiding sending the context across the network to a different node. This could favor an all-around chip which both has super memory thruput and really efficient computation.

4. Flow through an MoE prefill layer

Classic prefill begins with attention. A block of new query tokens attends to reusable prefixes like the system prompt and to earlier positions in the new block. The attention operation produces one activation for every new token, and shared transformations prepare those activations for the routed-expert stage.

The router assigns several experts to every token. If the model selects eight routed experts, each incoming token produces roughly eight routes across a bank that may contain 256 experts or more. In token order, those records are sparse and interleaved: neighboring tokens can have completely different destinations. Across the full input block, however, every expert can accumulate a short list of assigned tokens.

The worker sorts or queues the route records by expert identifier. All activations assigned to expert 037 become one message stream; those assigned to 142 become

another. A single expert-weight load can then serve all activations in that bucket. Several requests can contribute to the same buckets, and separate attention instances running the same layer can merge their routed work at the expert destination. Converting token order into expert order is one of the central efficiencies of MoE prefill. Scale-up networks like NVLink can be organized as memory mapped connections so sending requests to specific experts could be mapped to pushing the message into a specific memory mapped location which is configured as a hardware queue. There is very little start and stop time. The experts can listen with the equivalent of an RDMA CIQ that again leverages hardware acceleration to deliver the message stream without need for complex protocol start and stop. Simple headers inline on the message identify the sender and receiver. A similar connection can be set up for the return flow from expert back to the decode worker.

The expert outputs are returned to token order, weighted, combined, and passed to the next layer. At the same time, attention creates new K and V state for the appended tokens. Across the model, those values form an immutable KV suffix which are used to decode the next token, and which are eventually stored as the next blob in the context for use by the next turn in the conversation.

Prefill therefore combines two forms of weight reuse. Attention and dense transforms reuse weights across many tokens, while routed experts reuse each selected expert across the tokens collected in its bucket. The gains eventually flatten as growth in activation traffic, KV traffic, and output handling take more time compared to the tensor calculations. Batching can delay time to first token, but a single request containing several thousand new tokens already provides substantial arithmetic intensity; prefill does not always need a large number of requests in each batch to run efficiently. Modern single-socket GPUs may be able to do the prefill even on a large model for a small batch in around a second.

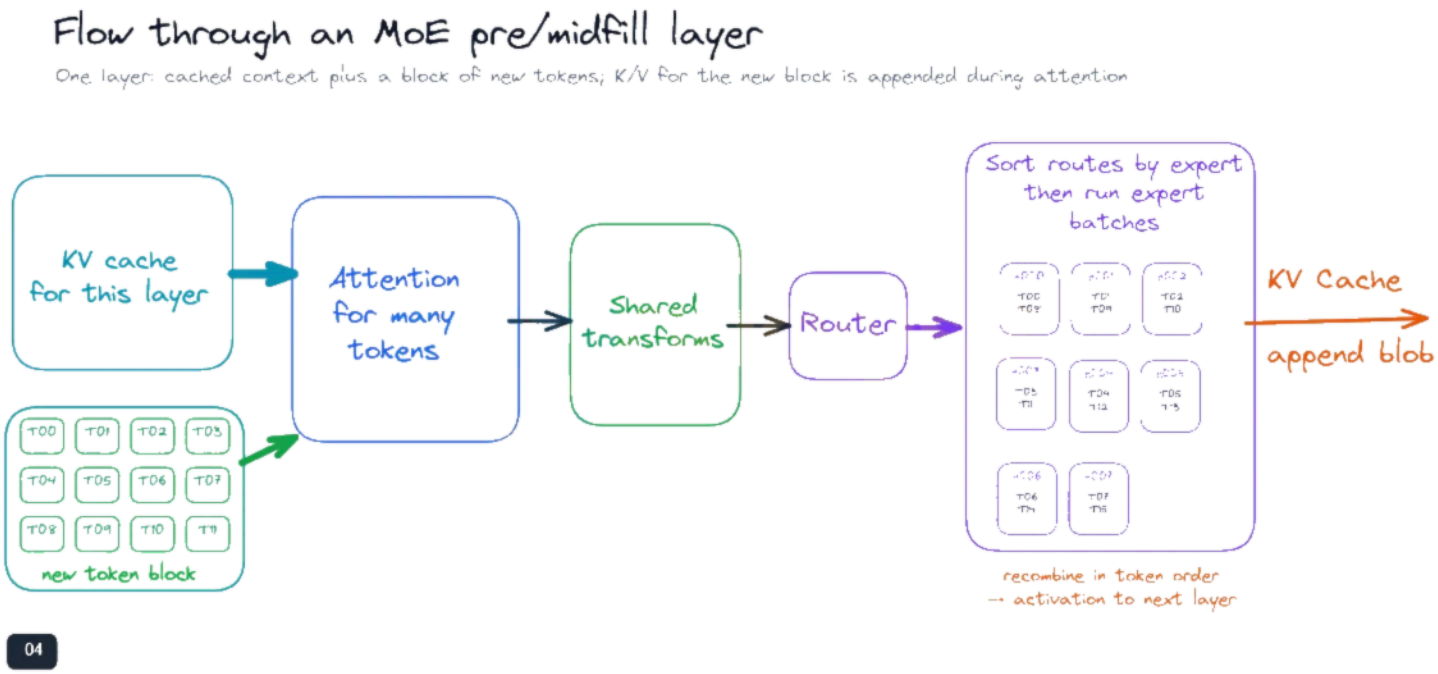


Figure 8. Flow through an MoE prefill layer.

5. Flow through an MoE decode layer

Decode begins with a much smaller amount of new work per forward pass. The current token, or a handful of speculated tokens in MTP (multi-token prediction) is the query. Attention reads the prior KV state together with the new token's KV entry. The new entry appends to the result and is appended to the KV state for the next pass.

For agentic contexts, the prior KV state is often a heavier read burden than the attention and expert weights used by that one token. The new query and its local projection weights are comparatively compact. Attention may be partitioned across KV heads, context ranges, memory channels, or accelerator units, after which partial results are reduced to one token activation. That activation is small enough to move cheaply even though it will trigger much larger local tensor operations in the next phase.

The activation passes through shared transformations and routing. The router selects a few experts from the available bank, and only those expert tensors become active for the current token. Their outputs are weighted, combined, projected, and passed onward.

At each layer, attention creates and retains that layer's K/V entry for the new token. The activation then advances to attention in the next layer, where the same sequence repeats. By the time the final layer produces the next token, every layer has appended its small K/V update. Those updates can be assembled into a new immutable KV suffix while the token is streamed to the client or agent.

The flow therefore contains both a broad operation and a sparse operation. Attention reads widely across private query state. Experts activate a small subset of total model capacity. The hardware mapping should support both without forcing the entire layer to use one parallelism strategy.

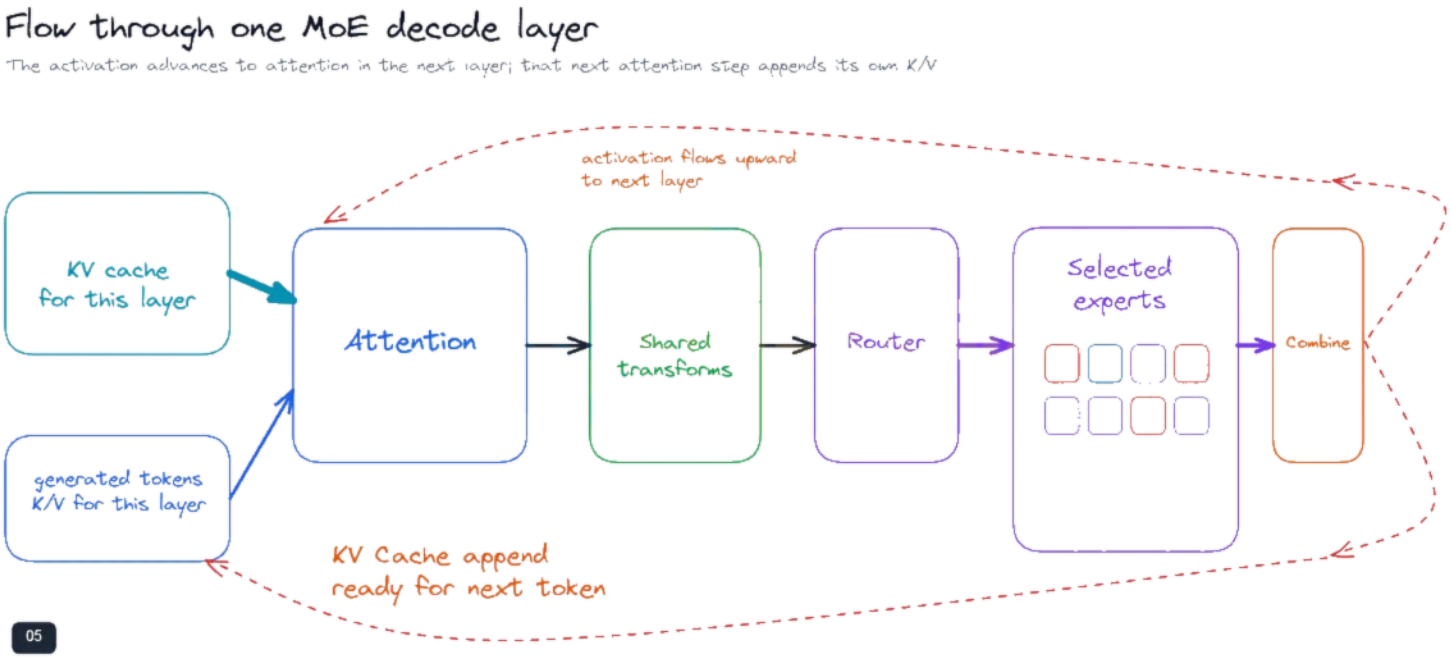


Figure 9. Flow through one MoE decode layer.

6. Midfill is a distinct operating regime

Midfill combines a long, cached prefix with a block of new tokens to prepare another turn in a conversation or agentic work. Traces show the cached input to be many times larger than the incremental query, although the incremental query typically varies from 50 to 5,000 new input tokens. The presence of cached system prompts, conversation roots, memories, prior agent steps, and retrieved documents can quickly raise conversational and agentic contexts to a million limit. AgentX datasets show frontier experts are repeatedly compressing contexts to stay under a million-token limit.

This makes midfill neither a small prefill nor a large decode. Like decode, it must read a substantial private KV cache. Like prefill, it processes enough new tokens to reuse weights and to sort routed activations into useful expert batches. Its arithmetic intensity can be hundreds of times greater than one-token decode, yet its cached-state traffic can be tens of times larger than that of a similarly sized initial prefill.

Modern accelerators can generally exploit the arithmetic reuse available in a block of hundreds of tokens, so the difficult balance is often between the large KV read and the

burst of routed-expert traffic. That burst does not match the smooth rhythm of a decode pipeline: a midfill can occupy a stage much longer than the decode batches in neighboring stages, leaving those stages underused.

The ideal parallelism can also differ. The worked example later in this essay finds midfill configurations that use different pipeline, tensor, and expert-parallel settings from decode. Reconfiguring a GPU in the middle of a live decode pipeline can be expensive or impractical. A service may therefore benefit from a dedicated pool of midfill-optimized nodes. The hardware can be identical to the decode pool while the loaded model stages, parallelism, and batch policies differ.

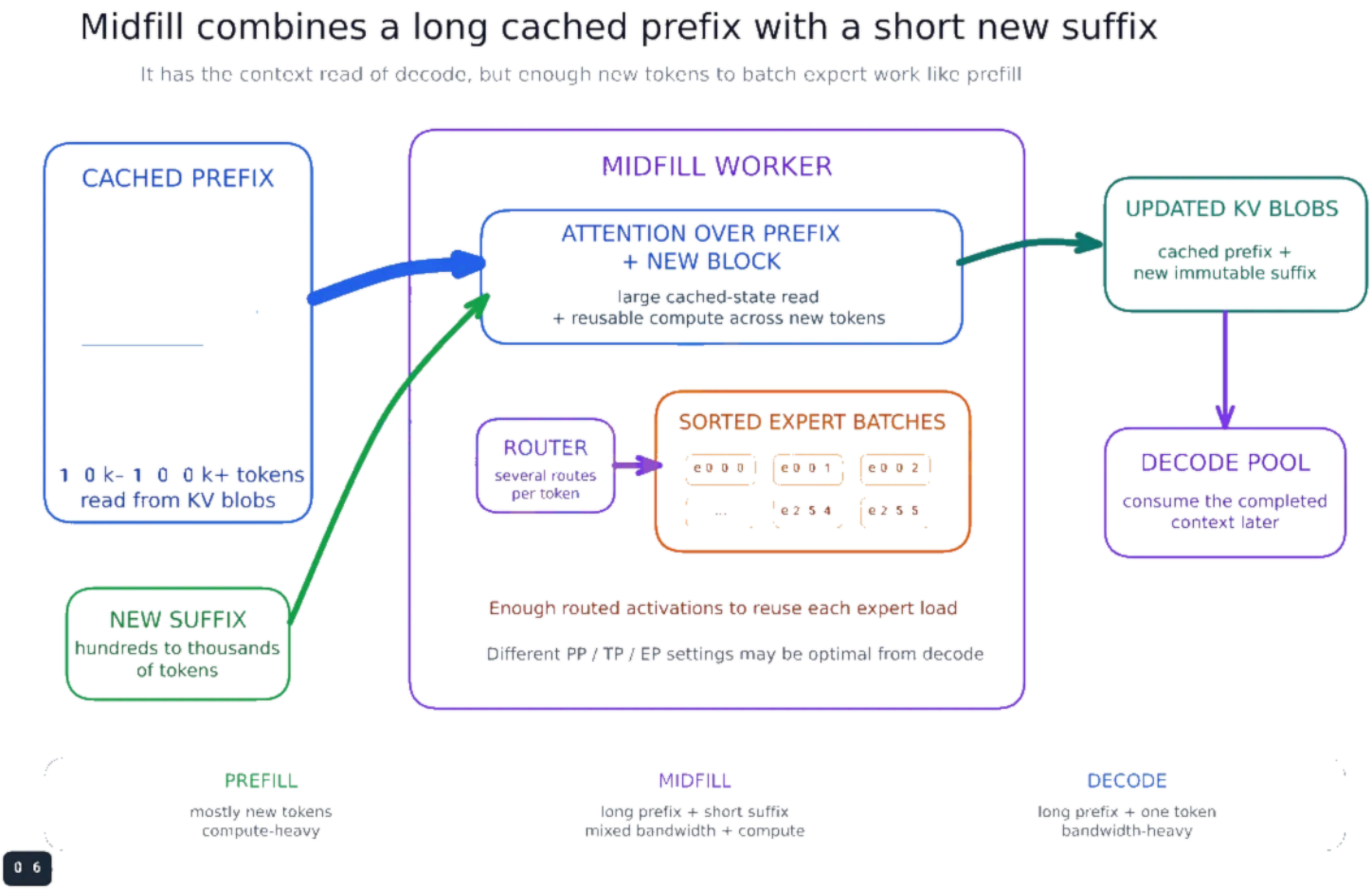


Figure 10. Midfill combines the cached-state read of decode with enough new tokens to batch model and expert work.

7. The model is a tall stack of repeating layer groups

A model's parameter count quantifies the total learned state, but serving depends on how that state is organized and used. The model is a stack of layers, and increasingly it is useful to group those layers into repeating architectural units.

In some MoE models, a layer group is the same as one layer. In others, a group contains one full-attention layer together with several linear, delta, top-k, or otherwise optimized-attention layers. Typically 1 to 3 of the initial layers may be dense or otherwise designed to get a clean start. The rest of the model will normally repeat one recurring group pattern.

A wide, shallow crêpe is a useful representation of a layer group. The broad surface provides room for attention, shared transforms, routing, expert capacity, and residual flow. Its shallow thickness marks the group as one slice in a much taller repeated model. It also echoes the physical form of an accelerator: a very thin active layer spread across a broad package. The crêpe is a useful intuition for mapping a wide algorithmic flow onto a wide hardware surface.

The repeated structure simplifies function planning. A mapping created for one group can be reused for the next. Corresponding memory regions can hold the next group's weights. The same compute and communication plan can execute repeatedly. Uniformity matters not only for runtime efficiency but also for compiler, kernel, and operations tooling.

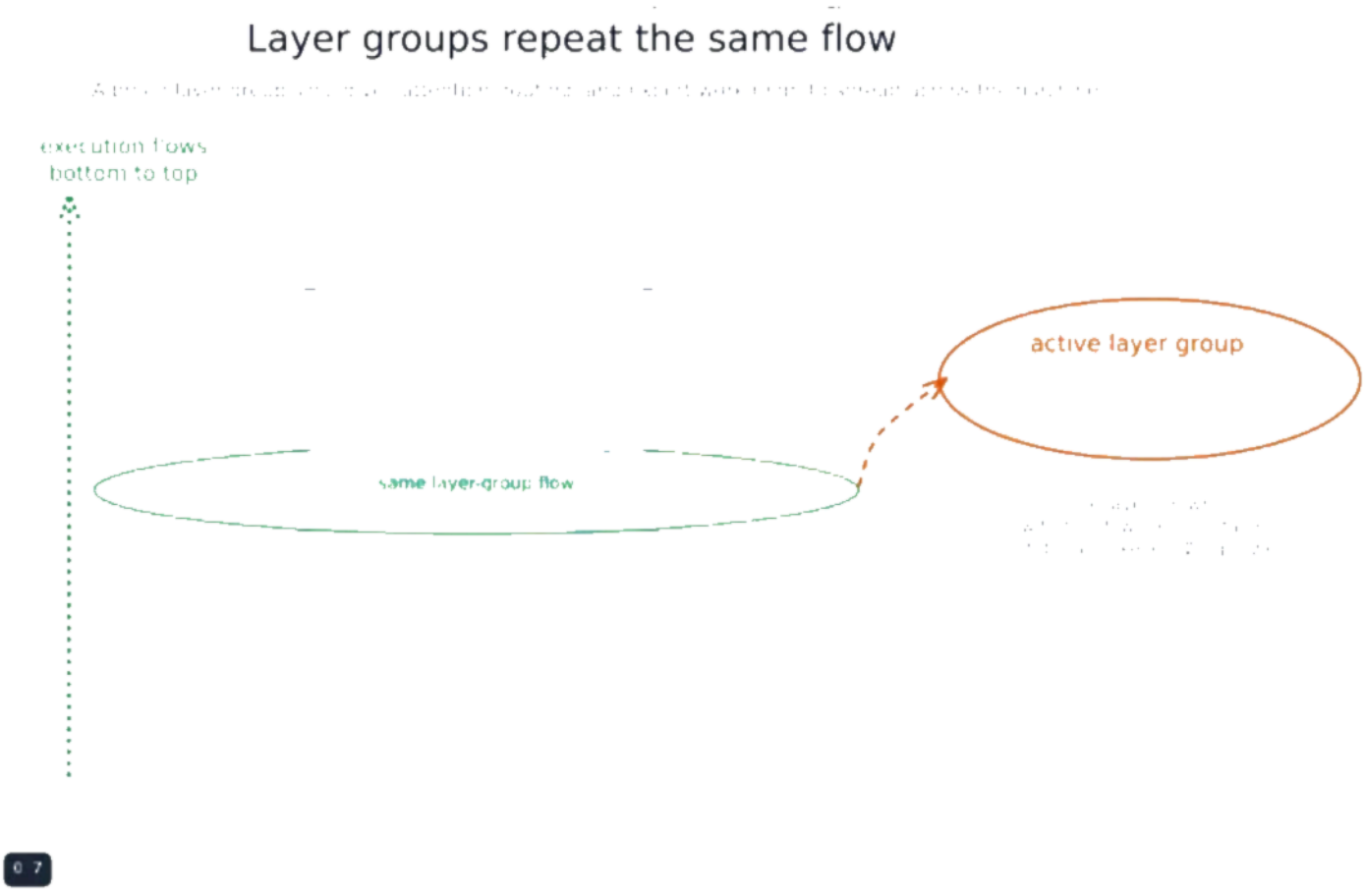


Figure 11. Layer groups repeat the same flow.

During decode, one layer is active at a time for a given token. Within that layer, several operations occur sequentially. Neighboring layers may be prefetched, and different pipeline stages may process different queries simultaneously, but the logical path of one token still climbs through the stack in order.

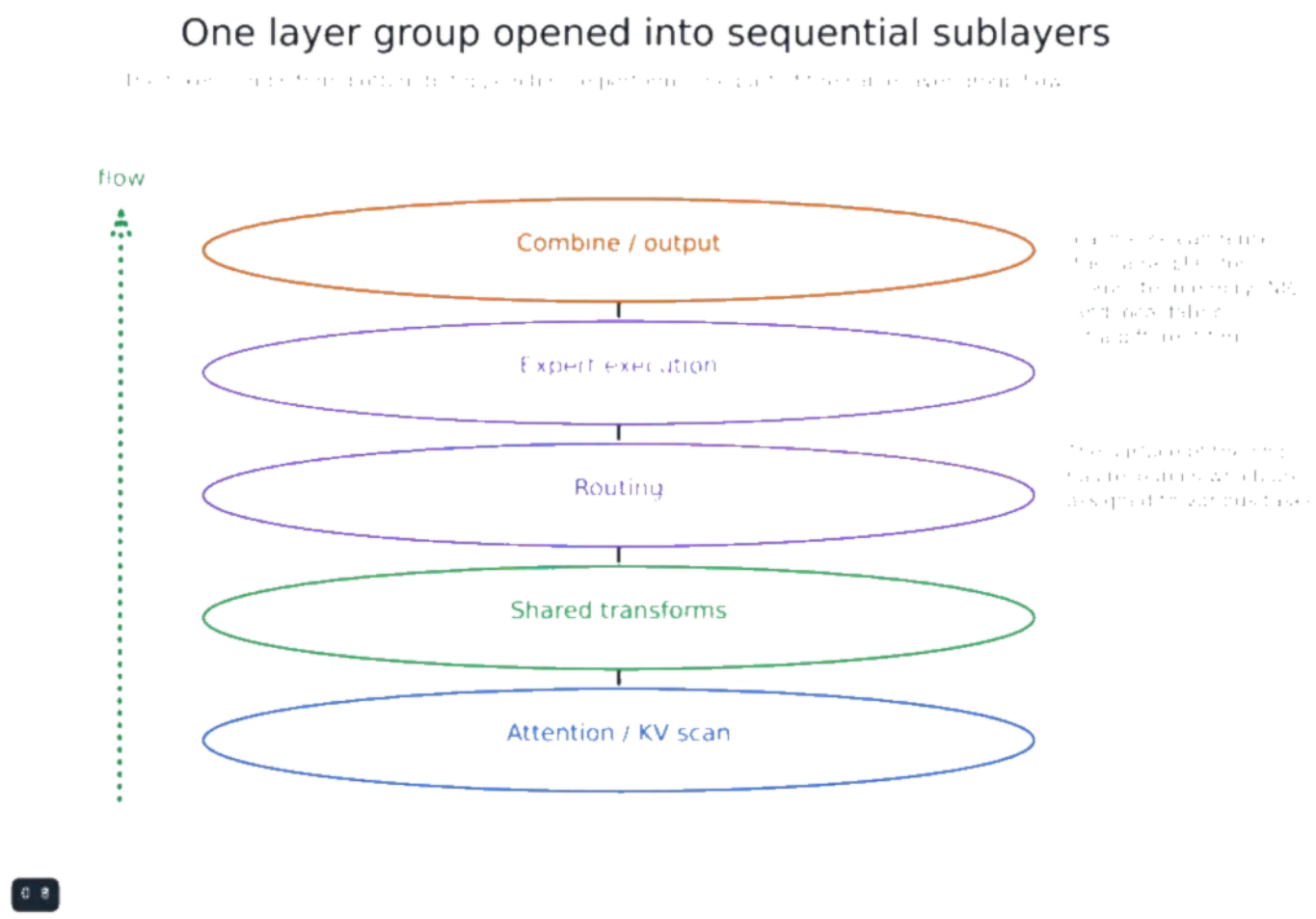


Figure 12. One layer group opened into sequential sublayers.

8. Parallelism should partition the narrow dataflows of the model

Parallelism is not one decision. Pipeline, tensor, and expert parallelism divide different dimensions of the model and should be judged by different communication patterns.

Pipeline parallelism follows the layer stack

Layers immediately suggest pipelining. The activation passed between layers is compact—often one embedded token, a small multi-token-prediction block, or brief summaries of prior hybrid group results —while the weights and KV state used inside a layer are much larger. A pipeline stage can therefore own one or more complete layer groups and pass a relatively small activation to the next stage without overpowered networking.

Pipelining divides model weights cleanly and keeps hot within-layer communication local. Its main drawback is multi-batch occupancy: a fully active pipeline has a different query or batch in every stage. Each stage therefore holds its share of the KV state for all live streams, and the memory benefit eventually stops improving as contexts grow.

Tensor parallelism belongs where one operation is intrinsically broad

Attention may need to divide weights and KV state across several accelerators. KV heads, context ranges, or another linear dimension can be partitioned so that each accelerator processes its share before a compact reduction of the token result. Tensor parallelism can also serve unusually large dense transforms.

TP should not spread automatically into every tensor of the layer. A collective that is sensible for a broad attention operation may cost more than the arithmetic of a small expert tensor.

Expert parallelism follows independent expert tensors

Most MoE parameter memory may reside in experts. Experts retain no query history of their own; context arrives in the activation. The expert bank can therefore be distributed widely across a peer set and shared by several worker instances.

Wide expert parallelism is useful because the experts are independent and numerous. The router sends compact activations to selected destinations, and those destinations return transformed activations. Expert-parallel width can remain large even if pipeline stages become smaller, allowing switch radix and local memory placement to be chosen independently.

The general rule is straightforward: use pipeline parallelism for the vertical layer sequence, tensor parallelism for large indivisible operations, and expert parallelism for the context-free routed-expert bank. A stage fits comfortably inside a scale-up domain when its available fast memory safely exceeds its local weights, live batch state, and working buffers:

$$M_{free,node} > \frac{W}{P} + M_{batch} + M_{working}$$

Modern GPU nodes often exceed that threshold by multiples. When they do, the demanding bandwidth, power, and latency requirements can remain inside one node while thinner pipeline activations and KV movement use scale-out links. The rest of the fast memory is probably not earning any revenue. Data sitting idle in memory is an expense, data moving to be processed is revenue.

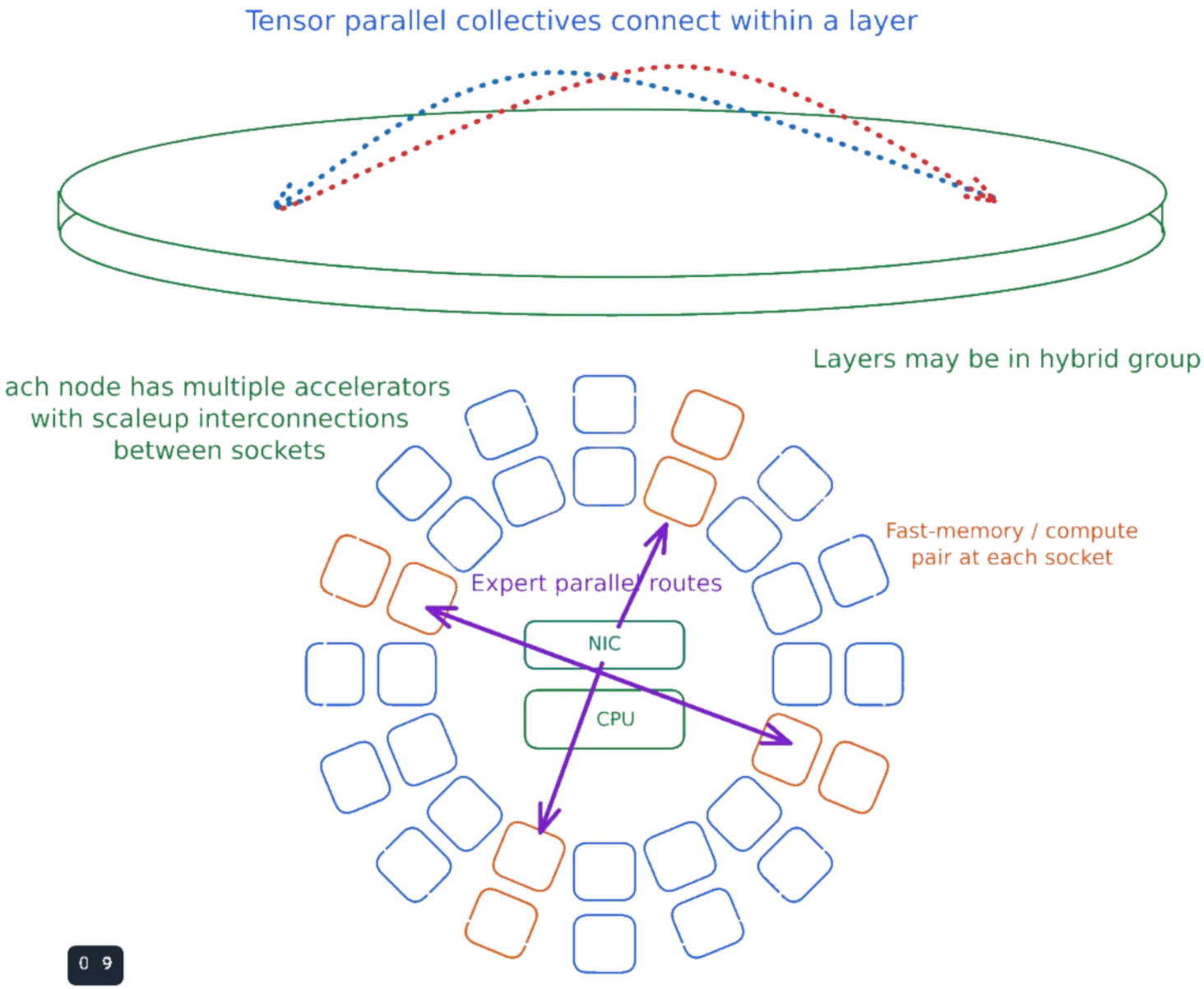


Figure 13. Pipeline, tensor, and expert parallelism follow different dimensions of a layer group.

9. A logical accelerator node

A logical node is easier to understand as a peer set arranged in a ring than as a literal circuit-board layout. The ring contains perhaps sixteen accelerators. Each accelerator pairs a compute region with local fast memory. The units connect through a strong scale-up fabric. A CPU coordinates workload and memory management, while a NIC connects the node to rack and datacenter resources.

The ring is just a visualization which shows the accelerators as peers, with equal standing. I am not implying the node-internal network should be a ring. It could be hub and spoke, rail, all-to-all, torus, hypercube .. whatever floats your boat, so long as it can handle the highest dataflows the rack contains with low latency and low energy per bit. I'll just draw it as a ring for simplicity.

This representation avoids committing too early to package placement, board routing, or switch implementation. The actual machine may use a central switch, several switches, direct links, or a hierarchical fabric. The logical requirement is a bounded peer set with predictable fast local communication.

The accelerator design may vary. One version may use SRAM-heavy processing-in-memory units. Another may use hybrid-bonded DRAM, or IGZO memory cells placed on top during BEOL processing. Another may use a hybrid-bonded high-bandwidth true 3D memory. Memory capacity and bandwidth can change while this logical model remains useful.

The node is the natural place to keep work that repeats at high frequency. Attention partitions exchange compact partial results. Routed expert activations travel to local destinations. Expert outputs return for recombination. Shared state and scheduling metadata remain near the CPU. The NIC carries pipeline activations, KV blobs, and work assignments beyond the node.

A node may be smaller than a rack like we see in an SGX node with 8 GPUs, or a scale-up system may span the whole rack like we see with NVL72. The logical symbol remains valid in either case. If scale-up already covers the rack, the separate rack-backbone level disappears and the next boundary is directly to scale-out networking.

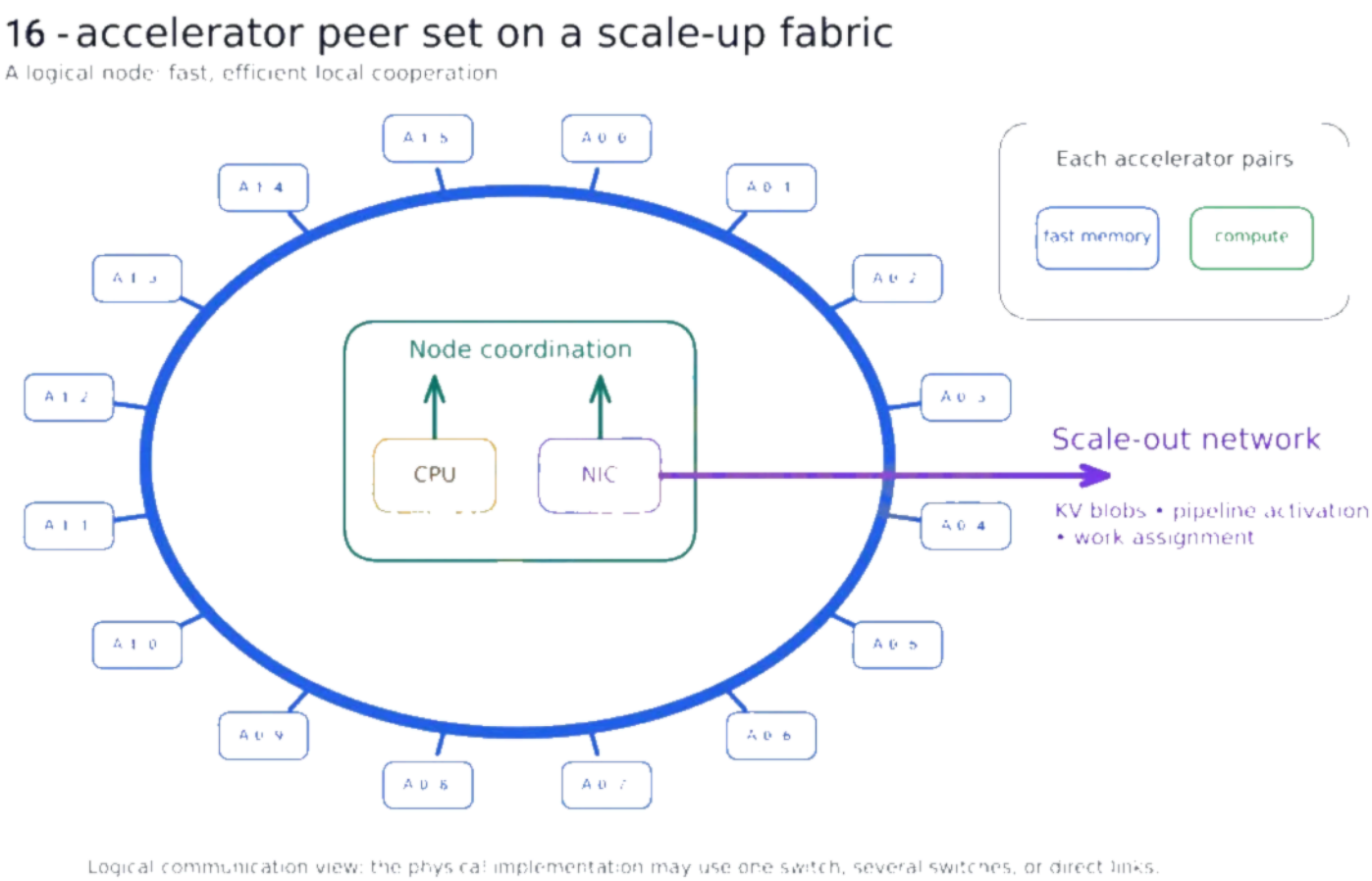


Figure 14. A 16-accelerator peer set on a scale-up fabric.

10. Nodes stack to make racks

The ring becomes more expressive when tilted as broad, shallow hardware crêpe. Several units can be stacked vertically in a rack, like model layer groups stacked in model depth.

This repeated form makes the model-to-hardware relationship easier to see. A layer-group crêpe can be placed onto one hardware crêpe. Several layer groups can occupy one node if memory permits. One layer group can span several tightly connected nodes if attention or expert capacity requires it. The mapping can expand or contract without changing the basic visual vocabulary.

A rack may be organized in several ways:

- one rack-wide scale-up domain;
- several node-scale islands connected by a rack backbone;
- pipeline stages distributed among those islands;
- wide expert placement within each stage;

- or a mixture of stage-local scale-up and shared scale-out links.

The rack backbone is therefore an optional middle level. Some systems extend the scaleup fabric across the rack needing no separate node connector. Others use strong local node fabrics and a distinct top-of-rack or shared-rail network. Above the rack, scale-out networking connects storage, prefill pools, decode pools, and other racks.

The repeated rack units also help separate two kinds of scale. The model can fit within one bounded rack worker, while the factory scales by adding many workers. The first problem is function placement. The second is worker orchestration.

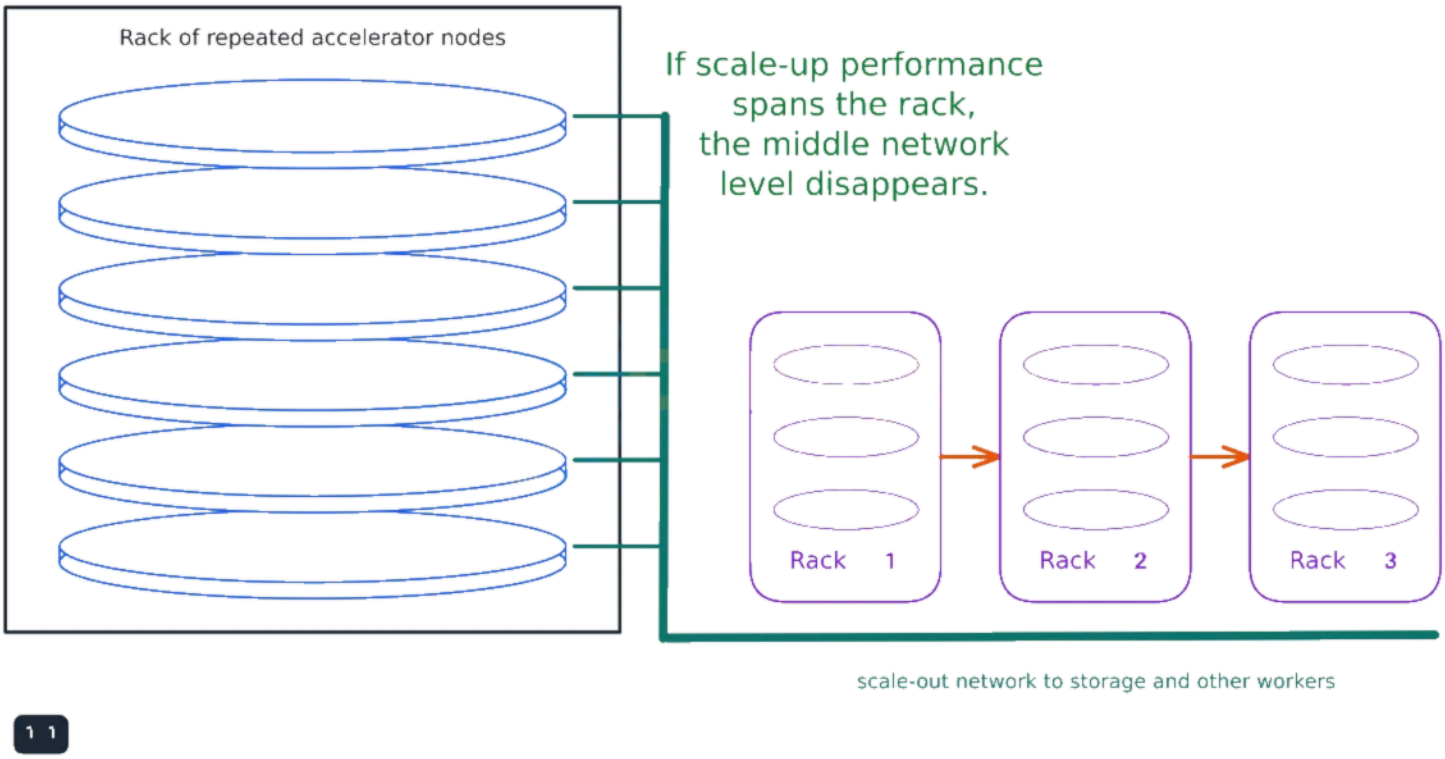


Figure 15. Accelerator nodes stacked into a rack.

11. One layer group reuses the node over time

A layer group does not require every operation in the algorithm at peak intensity simultaneously. Its operations form a repeating sequence that reuses the same hardware for different work over the decode cycle.

Attention may spread across much of the ring, using many memory channels and several compute units. The query and output transformations use model weights in more compact operations. Shared feed-forward work may use a dense region. Selected experts then activate local memory-and-compute regions around the ring. Recombination returns a compact result.

The same physical accelerators can participate differently at each step. Execution units that calculate attention may later execute one or more experts. Memory channels used to stream KV state can later feed expert weights. Buffers and local links are reused as the token advances.

This time-sharing is central to an efficient design. Static diagrams can make the machine appear underused because not every block is active at once. In reality the hardware is serving a sequence of different operations, and the goal is to keep the sequence moving with little setup or synchronization overhead.

The broad layer-group crêpe and the accelerator peer set are complementary. The crêpe shows the algorithmic flow; the ring shows the reusable physical resources beneath it. Mapping is the act of aligning each stage of the flow with a suitable subset of the ring.

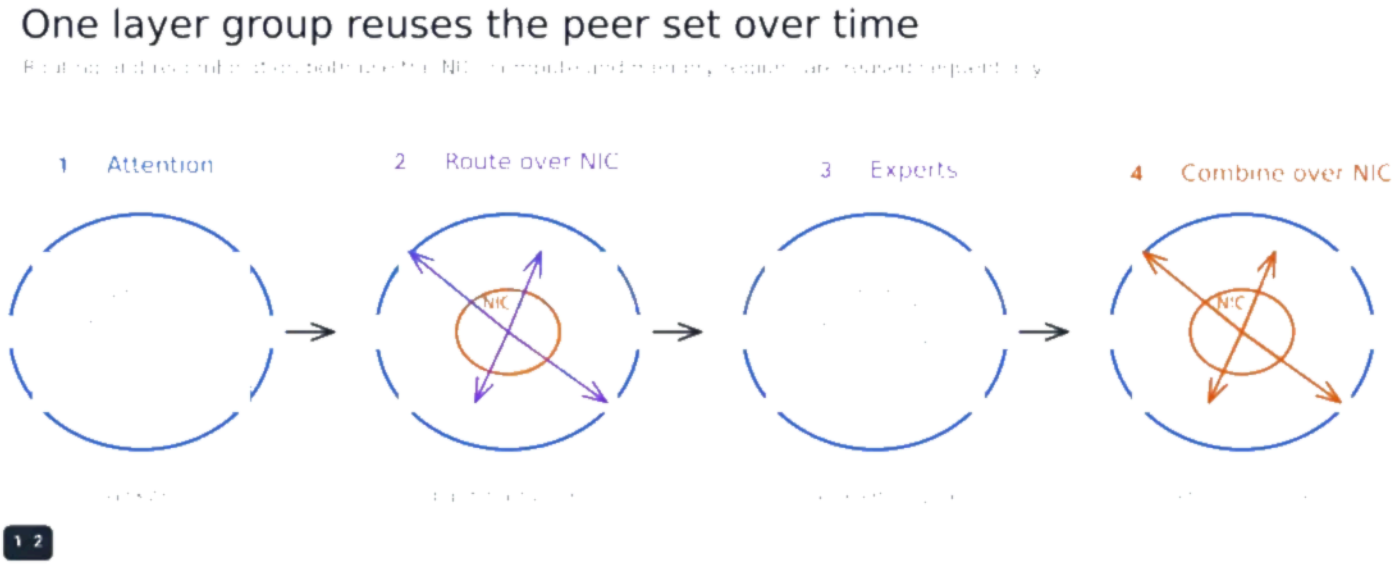


Figure 16. One layer group reuses the peer set over time.

12. Put throughput boundaries where the flow is smallest

Once the model and machine are drawn as connected flows, the placement rule becomes clear: frequent, latency-sensitive communication should remain inside the strongest local fabric, while weaker links should carry compact or amortized transfers.

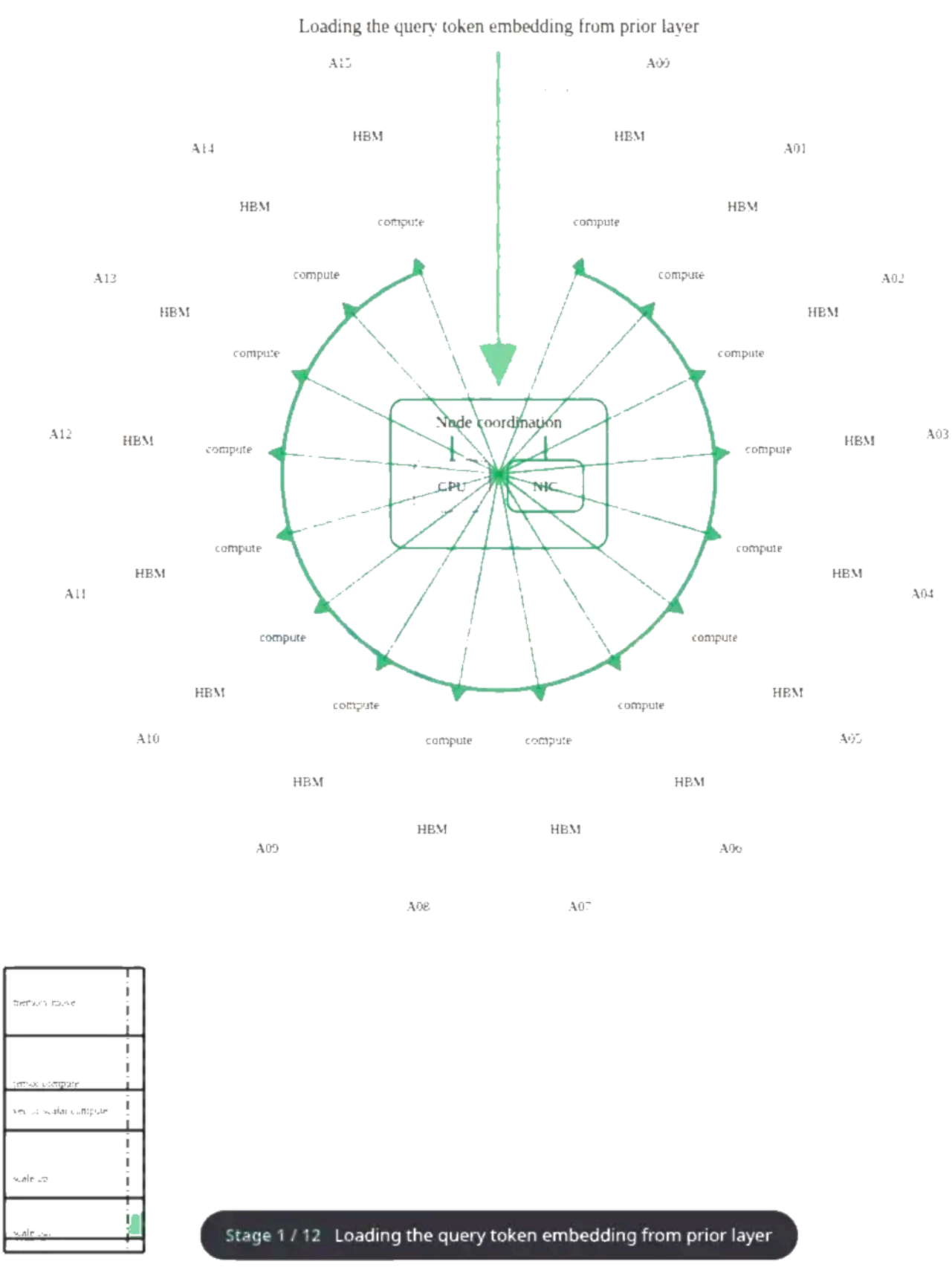


Figure 17. The general flow of operation of an MoE pipeline stage (animated).

Attention partitions may exchange partial results every layer. Expert routing sends activations to selected experts and returns their outputs. This traffic is not always the largest by byte count, but it is frequent, bursty, and sensitive to synchronization. It belongs inside the chosen scale-up domain whenever practical.

The activation passed between layers or layer groups is much smaller. A pipeline boundary can therefore cross a weaker link without moving the layer's full internal

working set. KV blobs are larger, but they move at worker boundaries and are amortized across many generated tokens. Their path can use striped or torrented scale-out networking and shared storage rather than consuming the innermost tensor fabric.

The router-to-expert boundary is typically a medium-throughput all-to-all carrying compact activations. Its difficulty often comes from endpoint count, synchronization, and routing efficiency rather than raw byte volume. Keeping it within the scale-up domain of a node or rack limits both latency and operational complexity.

This hierarchy is easier and cheaper to build than a machine in which every link is equally strong. Scale-up paths surround the operations that exchange embeddings and partial results within every layer. Scale-out links carry pipeline activations, KV movement, scheduling traffic, and traffic between bounded workers.

The tiers also have different latency requirements. A pipeline transfer between layer groups may barely notice a few microseconds if throughput is sufficient. By contrast, when an expert load and multiply take only a few microseconds, an additional 100 ns at each routing hop is material. The closest links should therefore be integrated tightly with the accelerator fabric, while less frequent transfers can tolerate the scale-out network.

13. Pipeline by layer group where practical

The layer group is the natural first candidate for pipeline placement. In a classic MoE model the group repeats one kind of layer. In newer structures it may contain one full-attention layer followed by several optimized- or localized-attention layers.

Keeping a group together can preserve internal dependencies and allow one placement and compilation plan to repeat across stages. Uniform plans reduce tooling cost and simplify deployment. Kernels, memory layouts, communication schedules, monitoring, and failure recovery all become easier when each stage runs the same recurring structure.

It is still possible to cut a pipeline in the middle of a group, although the resulting placement is more complex. If the layers have no special cross-layer dependency, the compact activation boundary remains a low-volume link suitable for networking. A

stage may contain several whole groups, one group, or part of a large group depending on memory and throughput requirements.

The guiding principle is not an absolute prohibition on splitting groups. It is to prefer regular boundaries and to understand when a nonuniform split is worthwhile. Tooling and operational simplicity can be as important as a small theoretical placement gain.

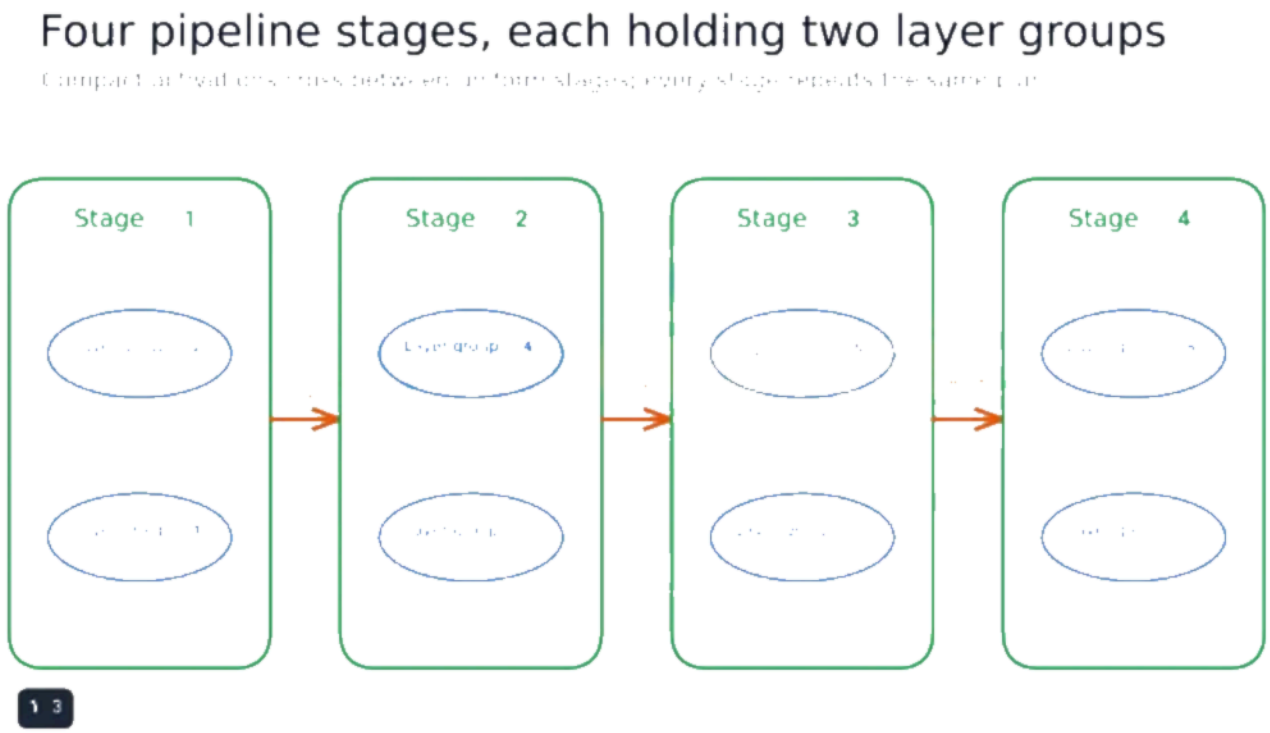


Figure 18. Four uniform pipeline stages.

14. Pipeline depth ends at the KV-cache crossover

Pipeline parallelism divides model weights cleanly. If a model has weight footprint (W) and uses (P) stages, the idealized weight footprint per stage approaches (W/P).

KV-cache demand behaves differently. A full pipeline remains occupied by running a different query or batch in every stage. In an eight-stage pipeline, each stage owns one eighth of the layer-wise KV values for each of eight live streams. The reduction in layers per stage is offset by the number of simultaneous streams, so the stage does not receive the same (1/P) reduction in live KV demand that it receives for weights.

A useful stage-memory estimate is:

$$M_{stage} \approx \frac{W}{P} + M_{KV,stage} + M_{working} + M_{slack}$$

Once weights per stage fall to the same order as KV and working-state memory, deeper pipelining produces little additional capacity benefit while adding coordination and latency.

The crossover depends strongly on model design. Classical full-attention approaches can require hundreds of kilobytes of KV state per token across all layers. MLA-like designs are commonly in the tens of kilobytes, while recent hybrid layer groups with mostly linear, recurrent, or local attention can be lower still. A working average near 25 kB per token may be reasonable for some new hybrids; approximately 70 kB per token is a conservative reference point for a proven compressed-attention design.

As a back-of-the-envelope design target, take 70 kB per token and two million tokens of aggregate live context assigned to one stage. That is roughly 140 GB of KV state. Double buffering allows one working set to move while another continues to run, raising the KV allowance toward 280 GB. Add model weights, activations, transient buffers, routing tables, fragmentation, and operating margin, and a stage target of roughly 400–500 GB of local fast memory becomes a useful 2027 benchmark.

Other designs can reduce requirements through streaming, compression, partial residency, or additional tiers. The estimate is not a universal requirement; it is a standard that a practical frontier system should be able to accommodate without heroic assumptions.

The stage can distribute this memory over several accelerator sockets if they share a scale-up fabric. More than 95% of the weight memory in a large MoE is in routed experts, which divide naturally among sockets. KV state can be partitioned by head, context, or attention instance, with the resulting collectives matched to the same local fabric.

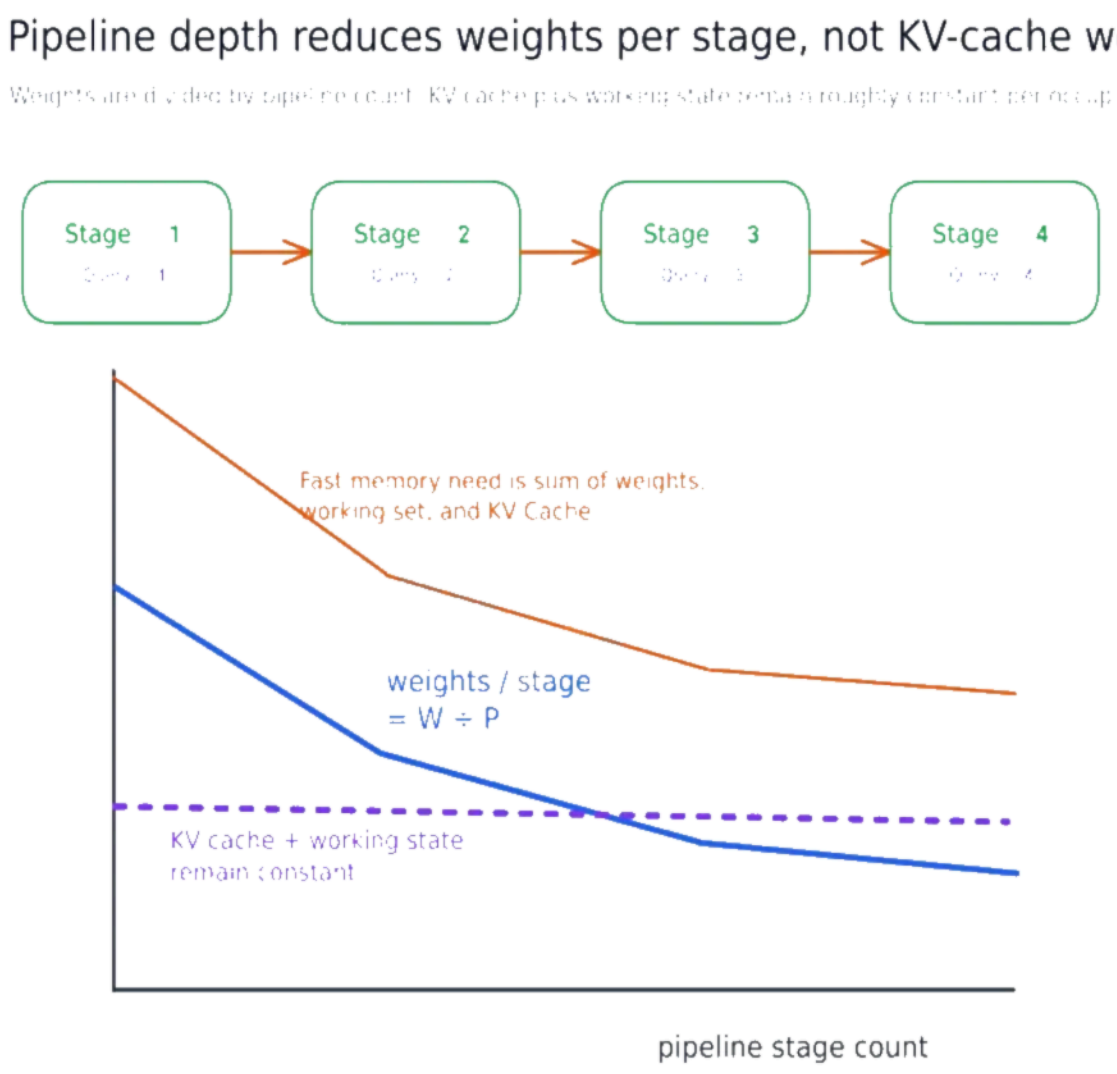


Figure 19. Fast-memory demand is stage weights, live KV state, working buffers, and margin.

15. Prefill and midfill turn sparse routes into dense expert batches

Prefill and midfill have strong batching opportunities because many input-sequence tokens complete attention before expert execution begins. The router can score all tokens, emit several destinations for each one, and then sort the route records by expert.

A practical input block can contain enough assignments that essentially every expert receives work. If 128 new tokens each select eight experts, the router emits 1,024 assignments across a 256-expert bank. Routing remains sparse for each token; sorting changes the execution order so that expert e000 processes all of its assigned tokens, then e001, and so on. Each expert is loaded once and reused for its compact token list instead of being repeatedly loaded in token order.

For an expert with (W) weights, weight traffic (B_W), activation traffic (B_A) per routed token, output traffic (B_O), and (n) assigned tokens, a rough arithmetic-intensity expression is:

$$AI_{expert} \approx \frac{2nW}{B_{W}+nB_{A}+B_{O}}$$

When weight traffic dominates, increasing (n) rapidly improves reuse. Eventually activation, output, and routing traffic become important and the gain flattens.

The sort is not free. Route records must be built, counted, permuted, sent to expert destinations, and restored to token order. Expert capacity limits can cause overflow or rerouting. Even so, prefill and midfill begin with enough tokens that the optimization is usually meaningful.

This is the key distinction from decode. The experts are identical; what changes is the number of routed activations available when the worker executes them.

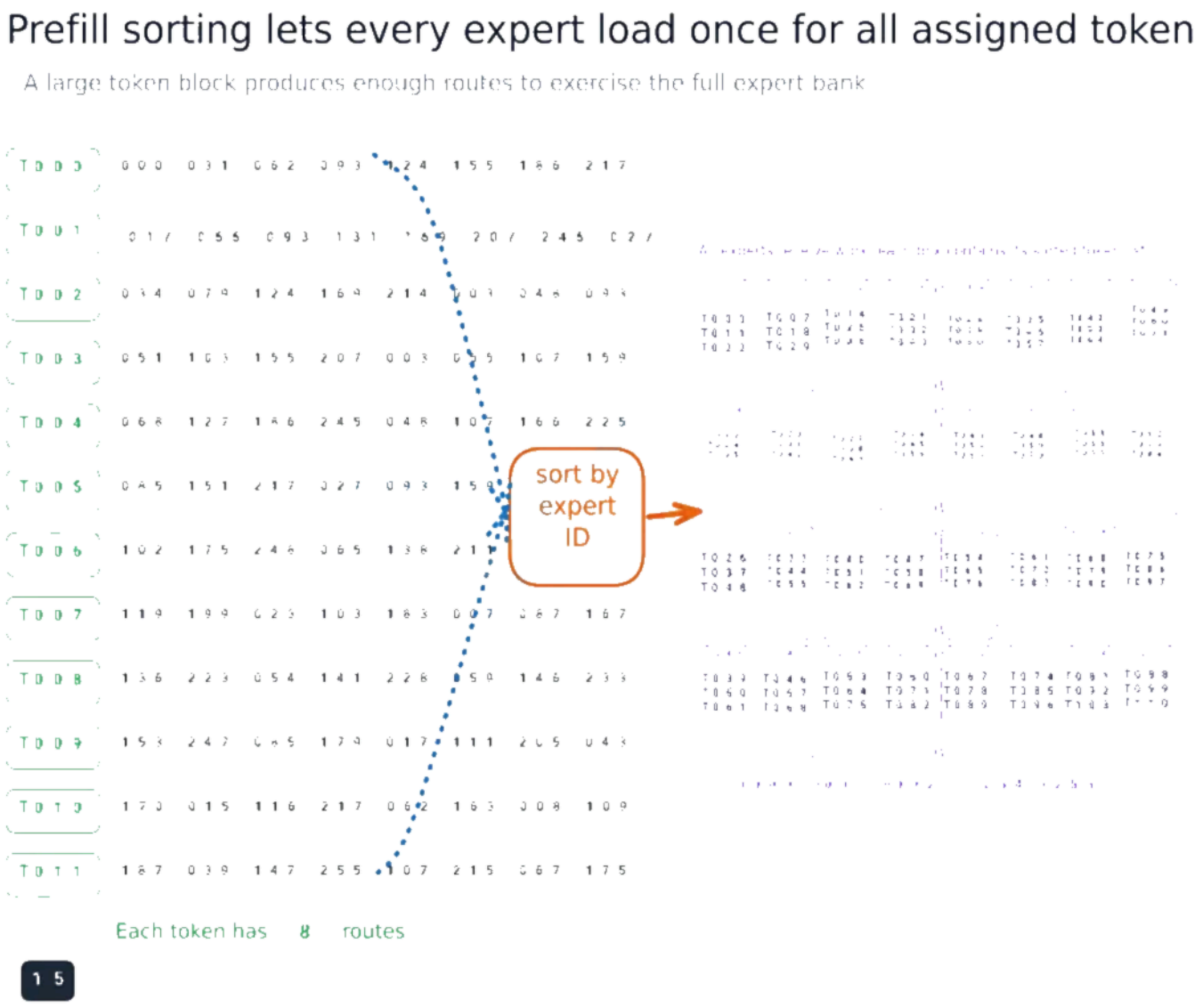


Figure 20. Sorting lets each expert load once and process every token assigned to it.

16. Decode gains less from batching

Decode attention receives little direct cross-query reuse. Each additional query contributes another KV cache and another context scan. Shared query-projection weights can be reused, and software overhead can be amortized, but the dominant long-context read remains private to the query.

Decode experts offer potential sharing because the expert weights are context-free. The practical limitation is sharing probability. A model may contain hundreds of routed experts in every layer and thousands of layer-experts across the model. Each query contributes one new token per pass and selects only a few destinations. At small batch sizes, only a few routed tokens are available and most active experts receive work from just one token. Some sharing occurs, but it grows gradually with the number of tokens in the pass.

Shared transformations provide more conventional weight reuse, but they are a smaller fraction of an MoE model. The overall throughput gain from enlarging a decode batch can therefore be modest compared with the increase in latency and KV-memory pressure.

A batch of one is not a trivial workload when context length is 100,000 tokens or more. The attention scan already occupies substantial memory bandwidth. The optimum batch size is not necessarily one—software launch, allocation, routing, and hardware utilization still matter—but it may be much closer to one or five than to the large batches associated with throughput-oriented serving.

This matters economically. Interactive tokens can be more valuable than bulk delayed tokens. Multiplying latency to gain a fractional throughput improvement may be a poor trade. Hardware and software should remove impediments to useful small-batch operation rather than making large batches a prerequisite for efficiency.

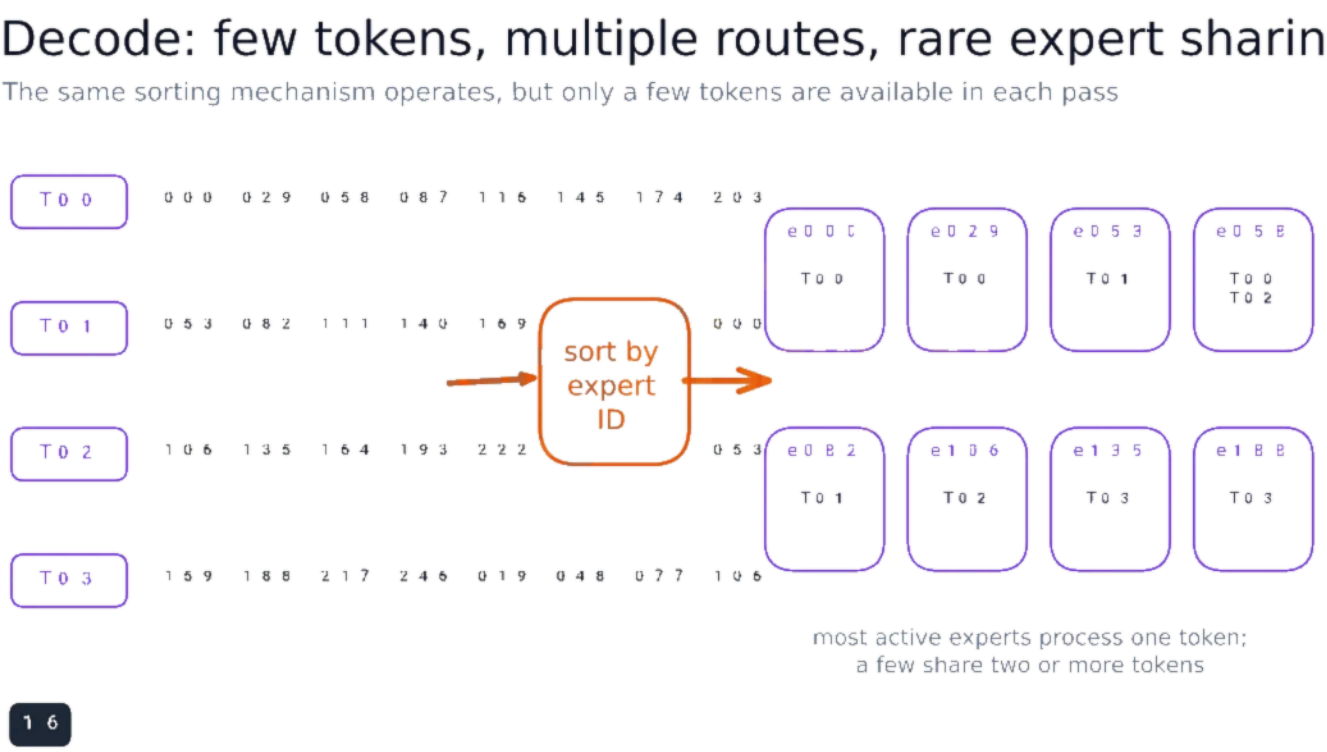


Figure 21. Decode uses multiple routes but rarely shares experts at small batch sizes.

17. Scheduling makes disaggregation useful

Disaggregation becomes most effective as a statistical system at cloud scale. A large pool can contain workers configured for long prefill, short prefill, long-context decode, short-context decode, or other useful classes. The orchestrator assigns each job to a suitable available worker and moves the required KV blobs through shared storage.

Prefill and midfill work are predictable once the prefix size, new-token count, and worker configuration are known. Decode is stochastic because output length is uncertain. Storage between the pools decouples predictable production from irregular consumption. A ready buffer of completed prefill and midfill work can keep decode workers fed even as individual queries finish at different times.

The scheduler should observe customer workloads and time-of-day patterns, anticipate known ramps, and reconfigure workers as the mix changes. Reconfiguration may mean loading different model stages, changing pipeline cardinality, reallocating expert capacity, or moving a rack among prefill, midfill, and decode.

Scale matters. In a small fixed benchmark, one input-fill stage may be connected directly to one decode stage. A slow stage can strand capacity elsewhere in the other stage. In a large pool, workers complete and start independently so no worker need be stranded.

High interactivity and high utilization are therefore not inherently opposed. The scheduler can keep batches small while keeping machines busy, provided it has enough workers, enough storage bandwidth, enough visibility into the work, and fast enough response times to adapt to the moment.

The service overview in Figure 1 provides the corresponding system picture: four resource pools with the orchestrator spanning beneath them.

18. Feedback can turn a small imbalance into oscillation

A tightly coupled system can amplify ordinary variation. Suppose admission to input fill is gated directly by the retirement of decode queries. A long-running decode query delays retirement. Input-fill admission falls. Decode later drains its ready queue and becomes underfed. Admission then opens aggressively, producing a burst of input-fill jobs and a new wave of congestion.

This is a delayed feedback loop. The control signal arrives after the system state has changed. Large batch recovery can make the loop worse by increasing KV-memory pressure and response latency. The factory alternates between full and empty rather than operating near a steady point.

No single scheduler policy solves the problem. Several familiar control techniques help:

- use ready-work buffers between stages;
- regulate admission from smoothed backlog and predicted service rate rather than individual retirements;
- separate fast local controls from slower capacity controls;
- apply hysteresis to worker reconfiguration;
- reserve capacity for short interactive work;
- and avoid changing batch size more quickly than the system can observe its effect.

The same principle extends across timescales. Microsecond worker scheduling, millisecond token flow, second-scale buffers, minute-scale worker reconfiguration, and hour-scale demand planning should not all respond to the same noisy signal.

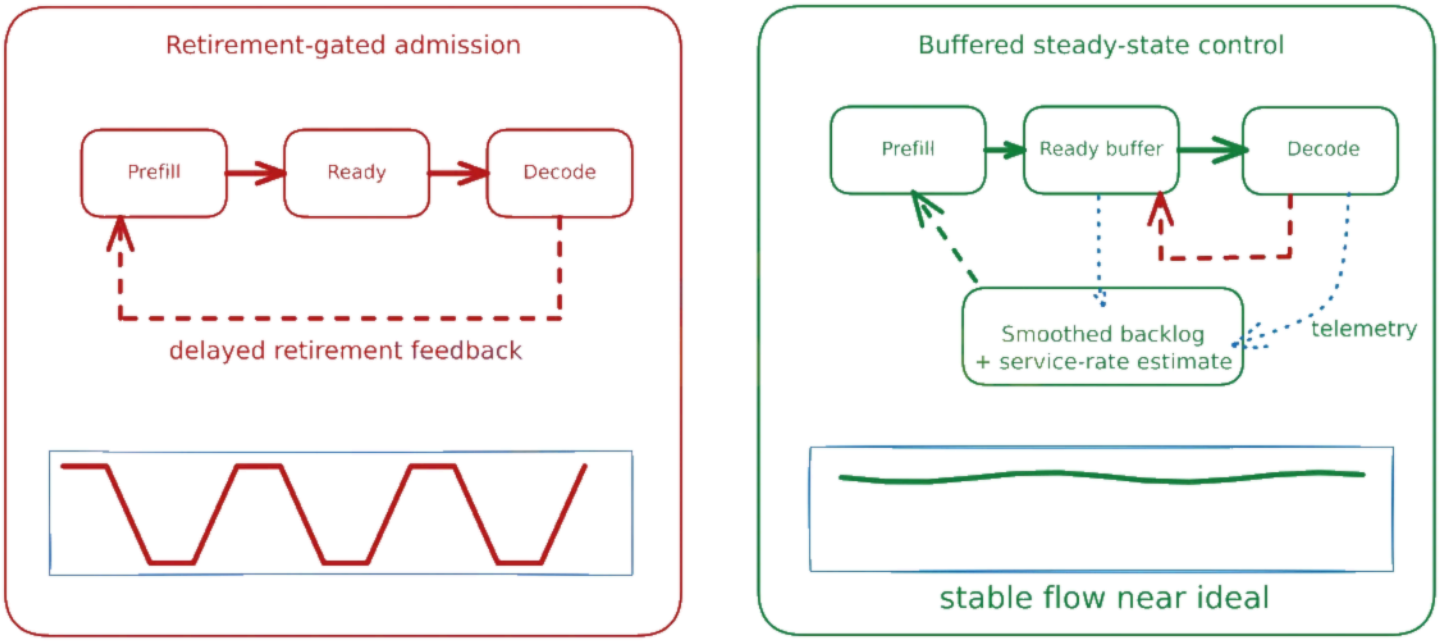


Figure 22. From retirement-gated oscillation to steady flow.

19. Aggregation Compared to Disaggregation

Now that we have exhaustively described disaggregated inference, is there a case to be made for aggregation? To some extent, yes. Clearly, since conversations can pause for minutes and hours it is not reasonable to keep a whole conversation tied to one system. That HBM can be put to better use within seconds of the end of a turn, and modern scaleout networks can move gigabytes of KV cache to a cheaper place in a parallel storage system, which can tier DRAM and SSD, in less than a second.

However, during a turn, do we need to schedule more than one place to get the work done? It is expensive to have too many different designs of server, adding operational costs as well as design work. What if there were only one design, and it centers on the midfill case? A request can be scheduled to that empty worker and stay there for prefill, midfill, and decode. It would need to have enough compute for prefill and midfill, and enough memory bandwidth for decode. There would always be some stranded capacity for either compute or memory transfer, depending on which stage the model is running, but we already tolerate low MFUs on GPUs. We try to overcome that with clever kernels, but in practice clusters run with stranded functions all the time.

Complex orchestration will result in some stranding. Some of that is avoided if a turn runs on a single aggregated worker. No schedule gap when pre/midfill hands over to decode. Natural sharing of one copy of the wide parallel set of experts. No network

overhead to move the context from fill to decode. The orchestrator still looks at the shape of the request – cache size, input sequence length, model type, customer interactivity level – and makes one decision which available worker should handle it through all stages of the turn. Then it leaves it alone until the stop token comes out and the turn is delivered to the user and cached to the fast parallel file system. The one worker uses kernels and dataflows adapted to the given request shape as it works through all stages of the turn.

Aggregated workers have their attractions and advocates.

All systems will always strand some capabilities of the accelerators in real world uses. In a disaggregated system there can be two (or more) types of machines built with investment focused on compute, memory, and network actually used, arguably the best use of that investment. Prefill machines might invest in compute, while decode machines might invest in memory. If workers are aggregated, then one machine will be used through all phases so it needs to invest in the strongest compute and the strongest memory thruput. Maybe also strong network. If is not an all-around standout, then it will be held back by its weaker sections and be an overall "meh". So, aggregation has advantages but it also leaves nowhere to hide in delivering the best of all-around performance. If you can build that magic all around star, then aggregating the whole turn in one worker could be best.

20. Memory bandwidth and memory capacity are different objectives

Memory is often discussed as two independent scalars to maximize. In practice, capacity and bandwidth trade against each other, and their economic value must be considered together.

Data earns revenue when it moves; it incurs cost while sitting idle. An accelerator with very high memory throughput can produce tokens at a rate that lower-bandwidth memory cannot match, even if the lower-bandwidth configuration has more capacity. Capacity is valuable until the active weights, KV state, activations, and operating slack fit cleanly. Beyond that point, additional local capacity may add cost but not revenue.

The capacity curve is not a vertical cliff. Below the useful limit, adding accelerators supplies both more memory and more bandwidth. The advantages end when other

limits appear: maximum scale-up width, poor model cuts, the dominance of latencies, extra synchronization, or network contention.

A safe-capacity design point must sit above those limits. It includes margin for realistic context distributions, double buffering, fragmentation, failures, and transient peaks. The revenue-capacity curve rises steeply before that point, rounds smoothly, and then reaches a flatter ceiling determined mainly by bandwidth and compute. Revenue may even decline if excess capacity becomes a sunk cost that operations try to justify with big slow batches of high throughput but inferior value.

Accelerator HBM is too expensive to hold idle data without immediate use. Completed KV blobs should move into CPU DRAM for outgoing staging and then into scale-out storage. Frequently reused common blobs may remain replicated in shared CPU memory or nearby cache appliances. Only the active working set can pay for the most expensive memory.

Tall stacks of high-capacity memory may impose costs in power, signal quality, packaging, and frequency. A design with just enough capacity and exceptional bandwidth may serve interactive inference at higher margins than one that maximizes capacity per accelerator with idle excess.

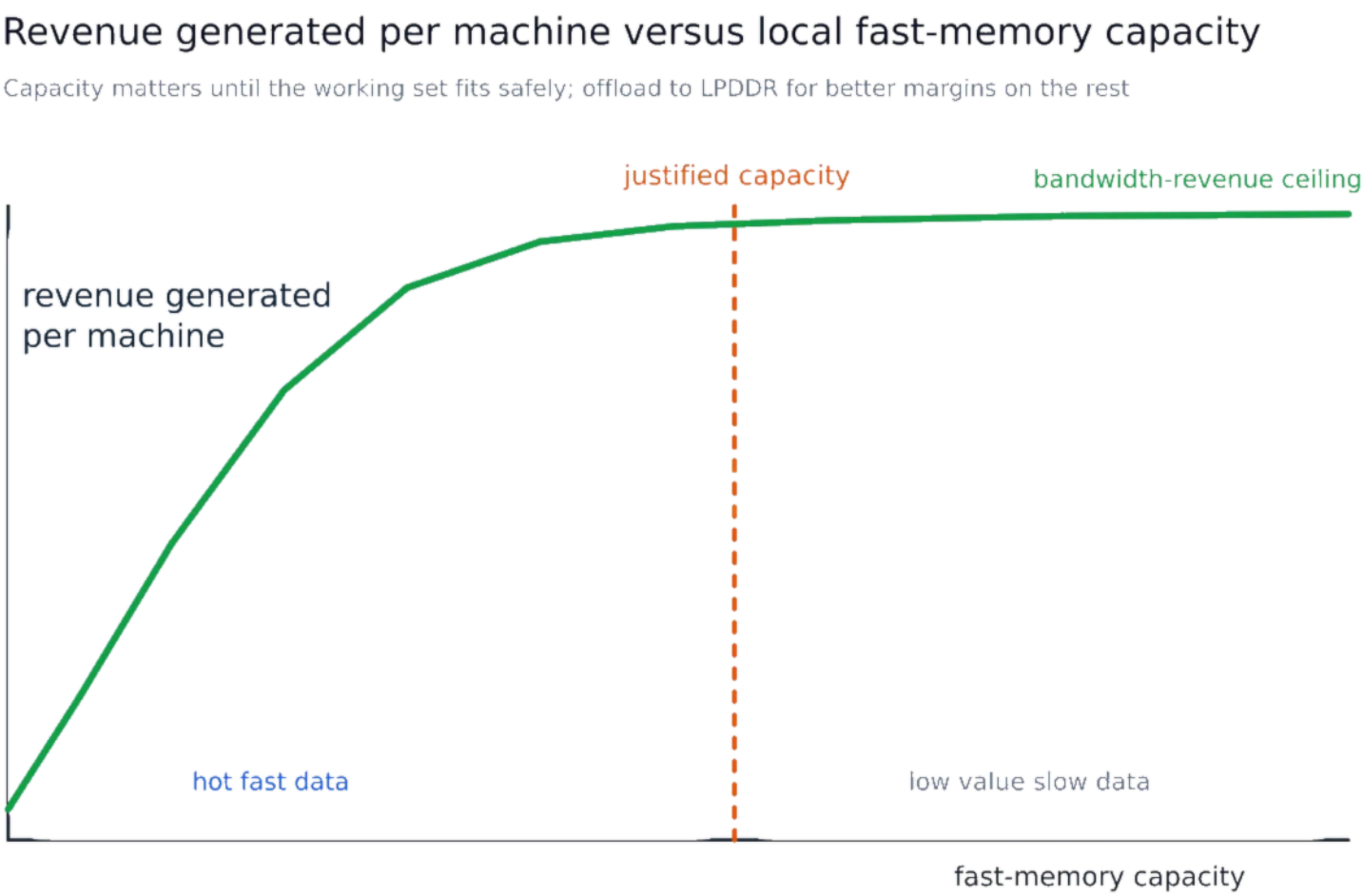


Figure 23. Revenue per machine versus local fast-memory capacity.

21. A practical mapping method

The prior sections can be condensed into a repeatable method for mapping a model onto a machine.

1. **Separate the operating regimes.** Measure prefill, midfill, decode attention, and decode experts independently. Each may split further by context length or workload class—for example agents, coding, media, or content creation. These become prepared configurations that the orchestrator can mix in production.
2. **Describe the repeating layer group.** Identify full and optimized attention, shared transforms, routed experts, and any cross-layer dependency.
3. **Measure active state rather than total parameters alone.** Separate stage weights, active experts, live activations, KV state, and transient buffers.
4. **Choose the natural parallel dimension for each operation.** Pipeline the layer stack, tensor-parallelize broad indivisible work, and distribute independent experts widely.

5. **Map the broadest operation first.** Long-context attention often determines memory striping and the minimum useful scale-up participation.
6. **Place expert weights in regular local destinations.** Let routing select among known peer locations rather than creating a global event for every token. A large model may contain on the order of 15,000 layer-experts, and each can be placed with regard to the other experts in its layer and the expected routing distribution. The goal is even use of memory capacity and bandwidth.
7. **Keep repeated high-throughput cooperation inside the strongest fabric.** Let thinner activations and amortized KV movement cross weaker boundaries. Respect latency and power cliffs as well as throughput cliffs.
8. **Stop increasing pipeline depth at the KV-memory crossover.** Once the model and target context fit with safe margin, additional stages bring little memory benefit and add stage-transition latency.
9. **Use input fill and decode differently.** Sort prefill and midfill routes into expert batches; preserve small-batch decode interactivity.
10. **Design storage and scheduling with the worker.** KV movement, admission control, ready buffers, and worker reconfiguration determine whether the physical mapping produces steady throughput. Short batches and continuous batch updates let the orchestrator replace individual requests without waiting for the whole batch to retire.

The method does not select one universal topology. It makes the trade-offs visible. A peer set may be one package, one board, or a rack-wide scale-up domain. A layer-group crêpe may map to one node or several. The important point is that the communication hierarchy follows the model's actual flow.

Map the flow before choosing the machine.

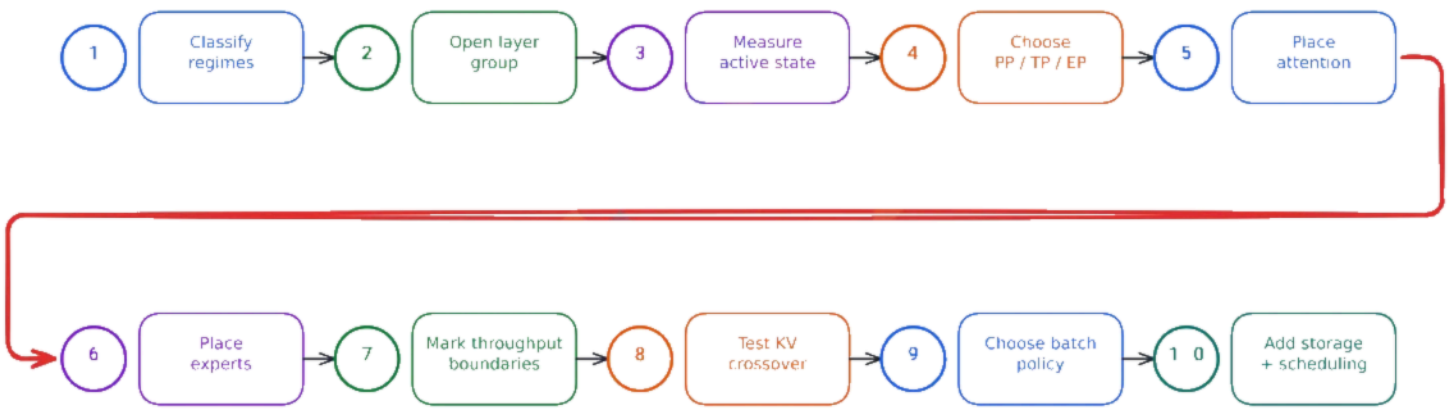


Figure 24. A practical sequence for mapping the model onto a worker and then into a token factory.

22. The next view is time

The diagrams so far describe structure and flow. The next step is to place time over them.

Interactive decode has a tight end-to-end budget. Execution of one layer commonly needs to fit within roughly 50 microseconds to 1 millisecond, which across a typical 60-layer model corresponds to roughly 15 to 300 generated tokens per second. Each layer contains several sequential phases, so individual arithmetic operations and handoffs live in tens of microseconds, microseconds, and sometimes fractions of a microsecond.

At that scale, tensor arithmetic is only part of the result. Kernel launch, queueing, synchronization, memory setup, routing, reduction, and link latency must all remain small compared with the operation they support. An expert multiply that takes a fraction of a microsecond is of diluted value if dispatch and data movement take several microseconds each.

The timing view also sharpens the network hierarchy. A high-bandwidth path may still be unsuitable if its setup latency is too large. A small local buffer can be more valuable than a large distant memory tier for a sub-microsecond handoff. A scheduler that creates millisecond variation can overwhelm careful microsecond engineering inside the worker.

The factory spans several control timescales:

- local arithmetic and link handoffs in fractions of a microsecond to microseconds;
- layer execution in roughly 50 microseconds to 1 millisecond;
- token generation in milliseconds;
- ready buffers and admission control in roughly 0.1 to 10 seconds;
- worker reconfiguration by the orchestrator over seconds to minutes;
- KV-blob retention and lifecycle in the fast parallel file system over minutes to days;
- and customer demand planning over hours and days.

These loops should be designed separately and joined carefully. Fast controls should not chase slow demand noise, and slow controls should not react to every transient queue fluctuation.

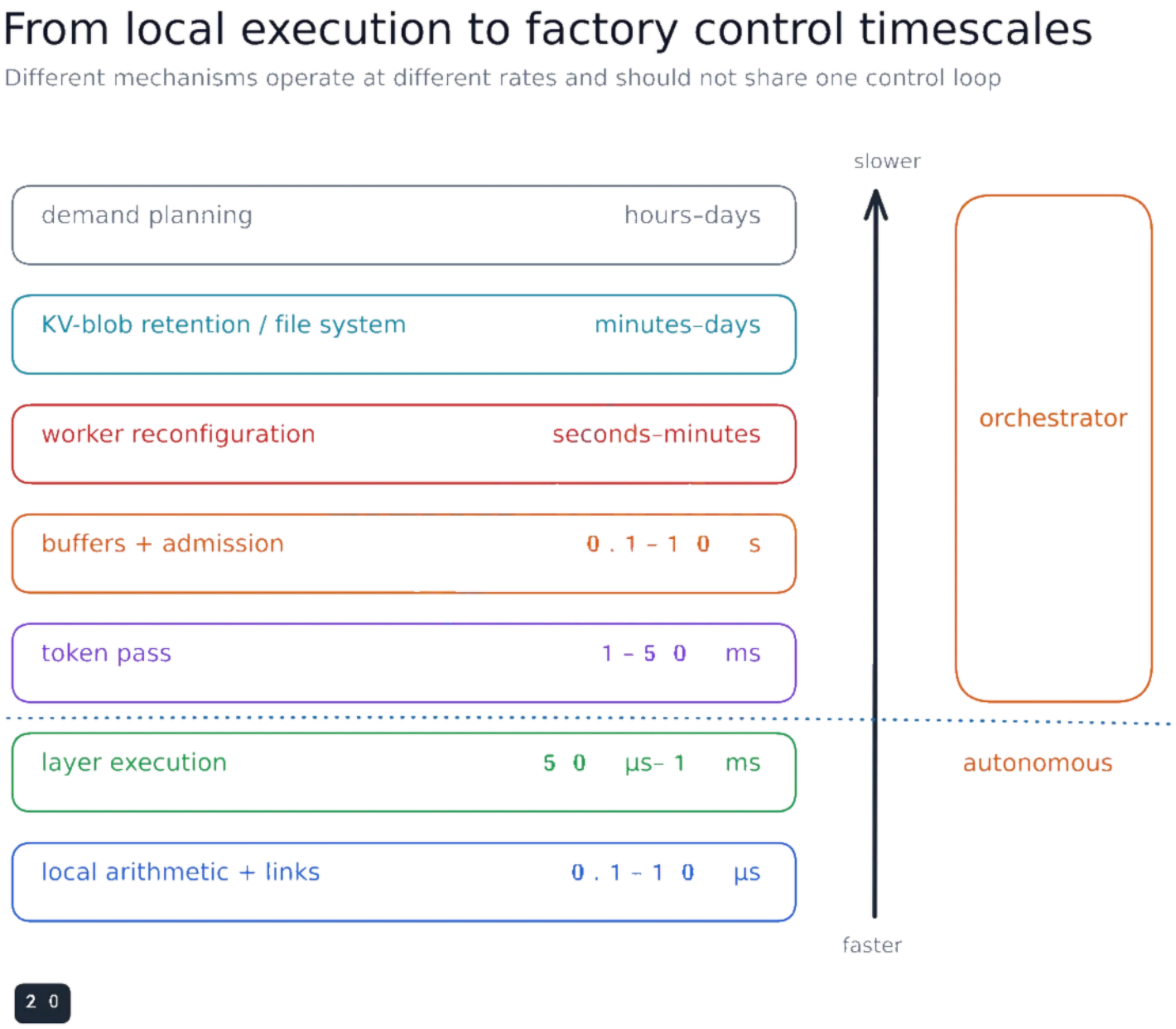


Figure 25. From local execution to factory-control timescales.

23. Worked example: projected Kimi K3 performance on Blackwell systems

The preceding sections describe a method for reasoning about model placement. A useful check is to apply it to a specific frontier model and explore the resulting trade-offs together. The following charts show modeled Kimi K3 performance on B200, B300, and GB200 systems using configurations from 16 to 64 GPUs in a rack. These are workload-and-hardware projections, not measured benchmark results. Connected lines identify projected Pareto frontiers: configurations for which interactivity cannot improve without giving up throughput or another selected objective.

22.1 Interactivity and throughput

The first chart compares time per output token for decode, or time to first token for prefill and midfill, against tokens per second per GPU. Normalizing throughput per GPU makes configurations of different sizes directly comparable.

The lower-left group contains decode frontiers for 8k and 32k input contexts. Feasible configurations appear for all three GPU types. GB200 is strong across much of the frontier, while B200 and B300 also offer competitive points. The lowest-latency decode configurations generally combine pipeline and tensor parallelism. At the throughput-oriented end, the highest tokens per second per GPU tend to come from one attention instance per GPU, without tensor or pipeline parallelism, while expert parallelism remains as wide as the system permits.

The upper-middle group is midfill: 127k cached input tokens followed by a 1k-token append before decode. GB200 leads the low-TTFT portion without pipeline parallelism, using TP=4 or TP=2. B300 catches up toward the throughput-oriented end. B200 and B300 generally use PP=4, with expert parallelism bounded within each stage so the expert all-to-all remains inside the node.

The two groups at upper right are 8k- and 32k-token prefills. All three GPU types converge on similar configurations: PP=4, with TP=4 or TP=2 favoring lower TTFT at some cost in throughput. Their close projected performance reflects the shared Blackwell compute architecture and the compute-limited nature of these prefill workloads.

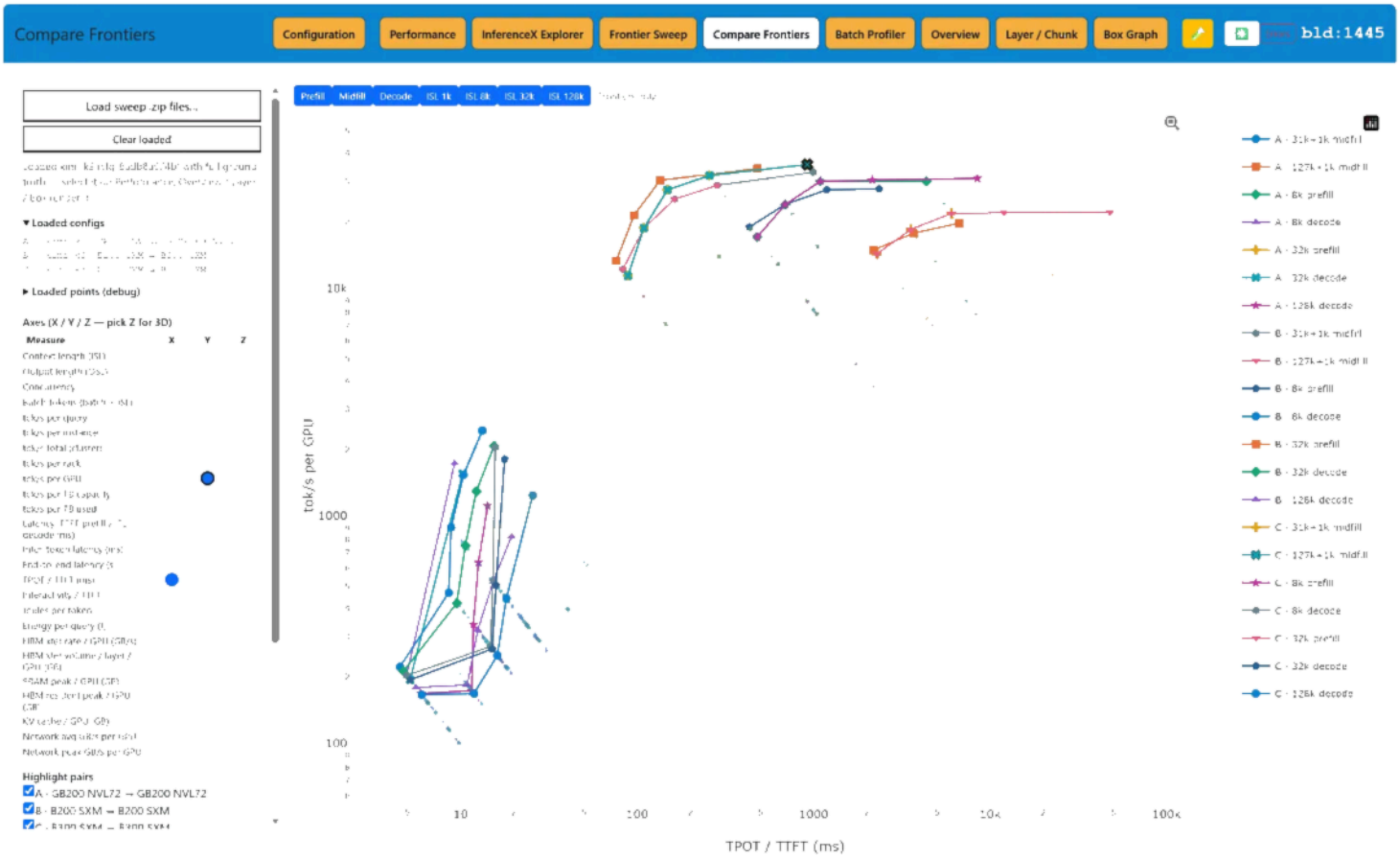


Figure 26. Projected interactivity and throughput frontiers for decode, midfill, and prefill. Source: SemiAnalysis Inference Simulator.

22.2 Energy per query

The second chart uses the same simulated configurations but changes the vertical axis to energy per completed query. Decode is the largest energy workload because the modeled output length is 1,000 tokens, requiring roughly 1,000 passes through the model for every query. Midfill uses the least energy: it appends a relatively short sequence after one pass through an existing long context. Full 8k and 32k prefills require successively more energy.

The model includes Kimi K3's localized-attention layers, so both retained state and computation reflect the expected savings from the attention structure rather than treating every layer as full attention.

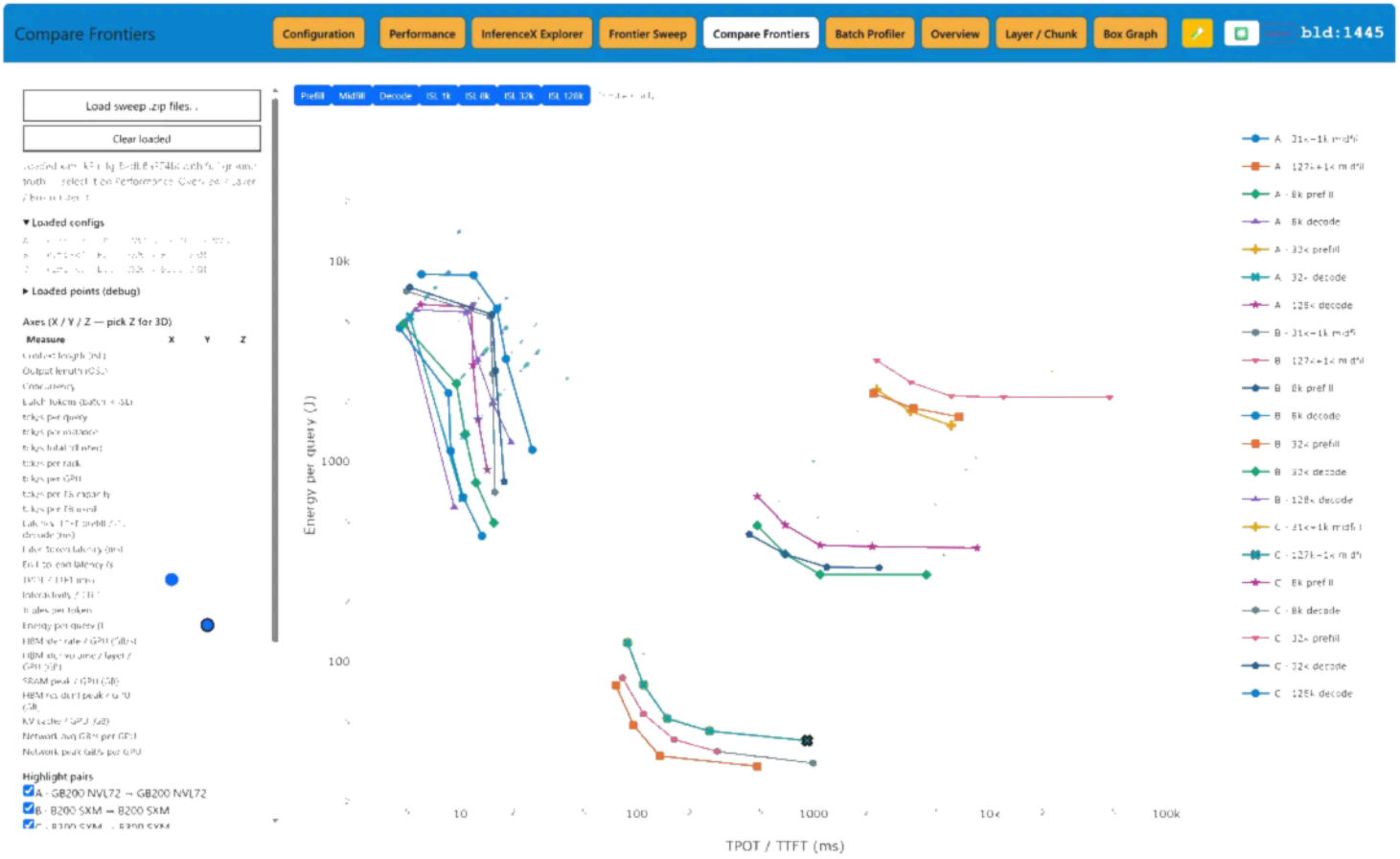


Figure 27. Projected energy per query for the same decode, midfill, and prefill frontiers. Source: SemiAnalysis Inference Simulator.

22.3 Fast-memory requirements

The third chart shows peak HBM residency per GPU. It excludes the extra buffers used while KV state enters or leaves the worker, which may add several gigabytes per GPU. Between conversational or agentic turns, inactive KV blobs can be transferred in parallel to network-attached DDR, including the DRAM attached to node CPUs. That memory is cheaper, more available, and less constrained by the HBM supply chain.

The projected frontiers cluster near the lower end of the feasible memory range. For prefill and midfill, the downward steps come from adding pipeline parallelism and increasing tensor or data parallelism, including data-parallel ring-batched instances. Decode reduces per-GPU residency first through tensor or data parallelism, using pipeline parallelism later.

Most frontier configurations remain below approximately 80 GB per G
allowing for practical KV-transfer buffers. This supports the earlier co
bandwidth can be more valuable than maximizing HBM capacity on ev
provided the broader storage and orchestration system moves idle state



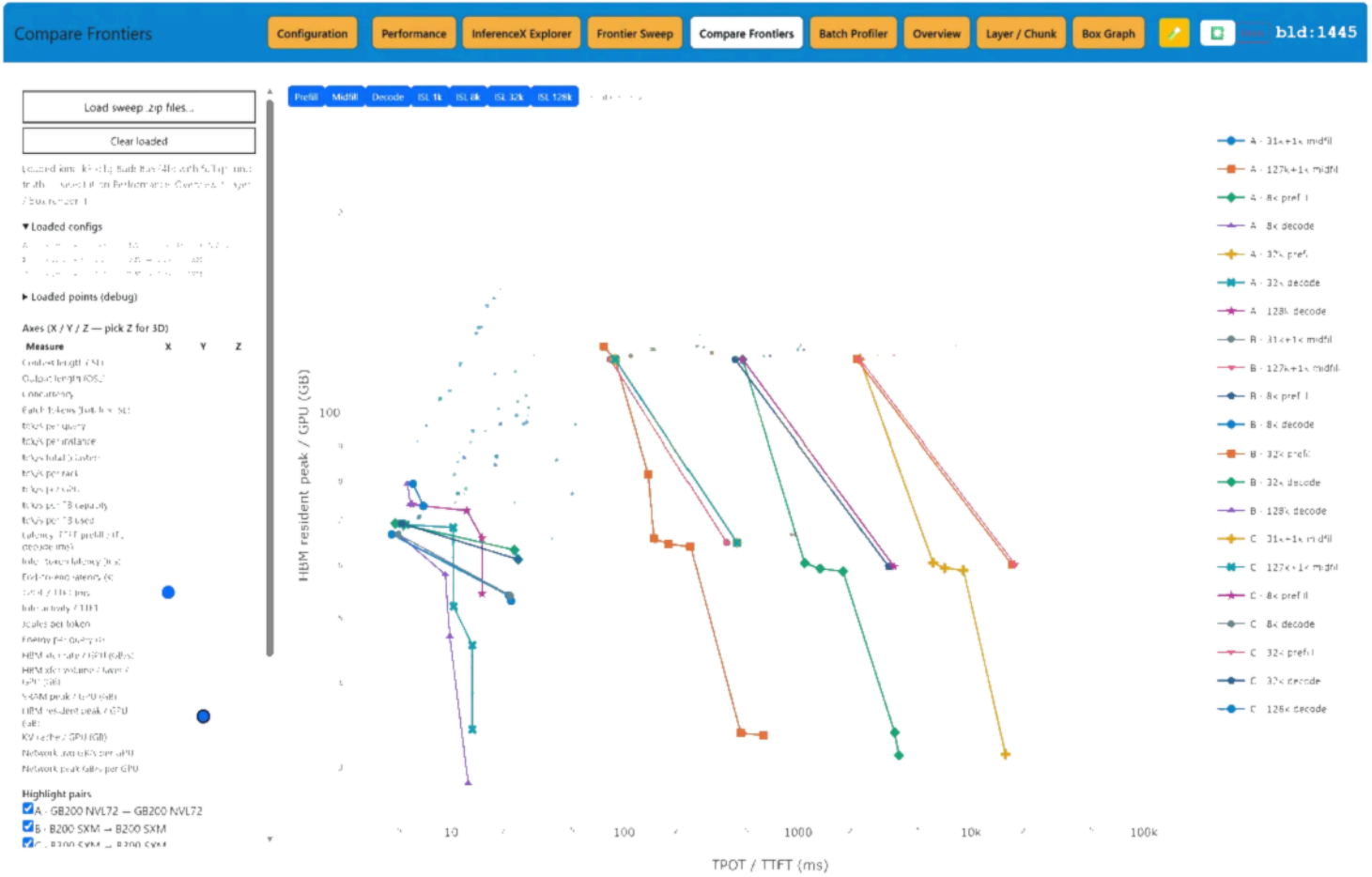


Figure 28. Projected peak HBM residency per GPU across the simulated frontiers.
Source: SemiAnalysis Inference Simulator.

22.4 Network peaks and the scale-up boundary

The fourth chart shows projected peak network traffic per GPU. Prefill and midfill make intensive use of the network for expert parallelism because many routed activations are in flight at once. Pipeline parallelism is therefore particularly useful on B200 and B300 systems: placing one stage inside a node keeps its expert all-to-all traffic on the local NVLink fabric.

GB200 NVL72 extends scale-up bandwidth across the rack. More GPUs can therefore participate in one layer in parallel, with all instances advancing through that layer in synchrony, to improve latency and throughput without crossing a weaker network boundary.

Decode shows lower interval-averaged peaks because a small batch consults relatively few experts at once. Individual transfers still burst at the physical NVLink rate; the lower plotted value reflects a measurement interval longer than the transfers themselves. This distinction between instantaneous link rate and sustained traffic matters when sizing the fabric and interpreting utilization.

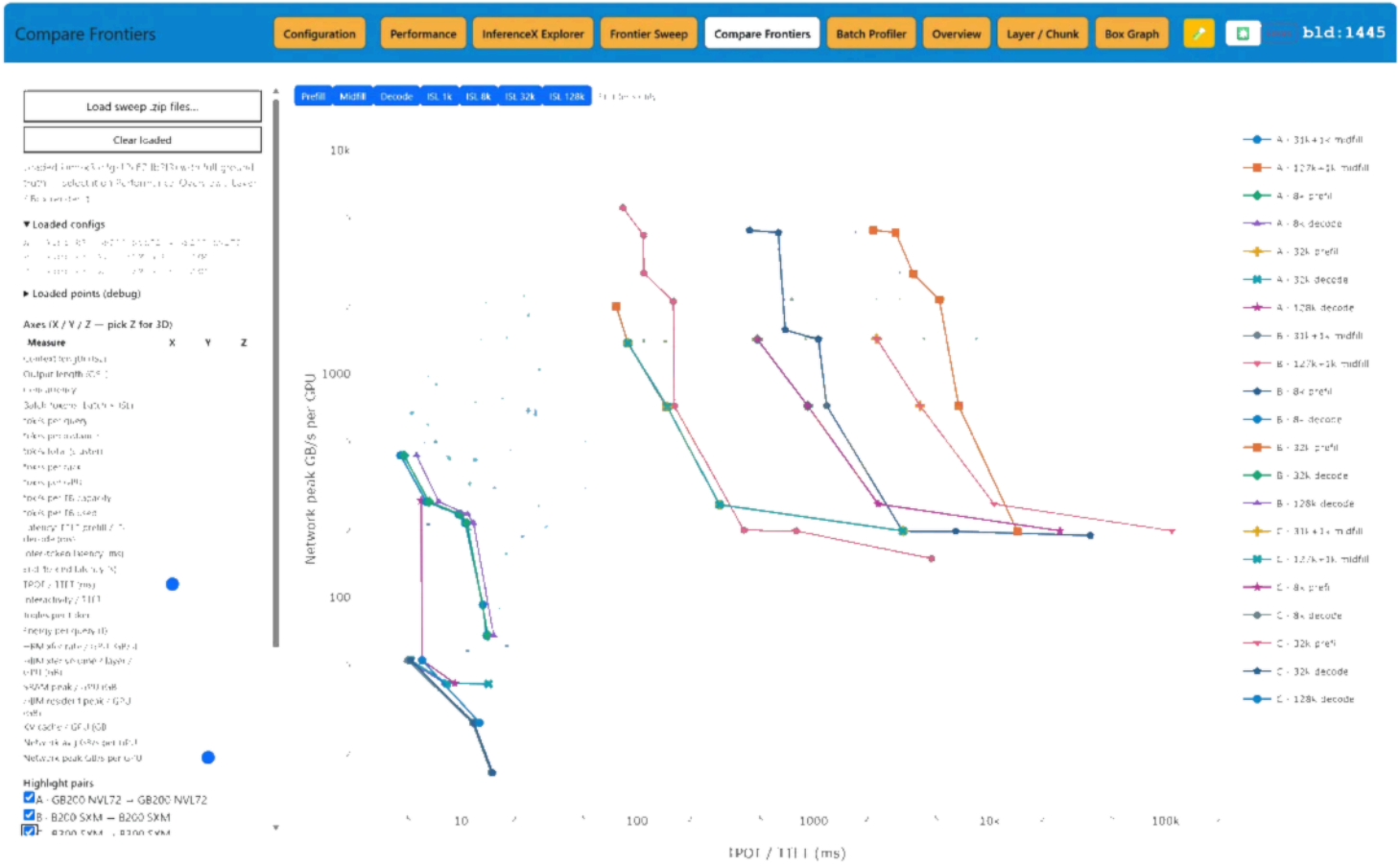


Figure 29. Projected peak network traffic per GPU and the benefit of keeping expert traffic inside the scale-up domain. Source: SemiAnalysis Inference Simulator.

The four views tell a consistent story. The best configuration changes with the operating regime and the selected point on the latency-throughput frontier. Prefill and midfill reward organized parallelism and stage-local expert traffic. Decode benefits from parallelism at the low-latency end, but the highest throughput per GPU comes from simpler attention instances combined with wide expert placement. Across the projected frontiers, fast-memory capacity appears less restrictive than network placement, memory bandwidth, and orchestration.

Closing perspective

The useful unit of analysis for inference is not total parameter count alone. It is the active flow of the model joined to the memory, communication, and timing hierarchy of the machine.

MoE provides structure. Layer groups repeat. Experts are numerous but individually manageable. Attention is broad and increasingly shaped by KV state. Operations occur in sequence and can reuse hardware. Pipeline, tensor, and expert parallelism can each follow the dimension where it is most natural. Prefill and midfill can sort routes into

efficient expert batches, while decode can preserve short batches and high interactivity.

The hardware can reflect that structure. A logical accelerator peer set provides a bounded scale-up domain. Nodes stack into racks. A rack may be one scale-up worker or several pipeline stages. Shared storage makes KV blobs movable. Datacenter networking connects bounded workers rather than carrying every inner tensor operation. Scheduling keeps predictable and stochastic stages in steady flow.

Holding the problem this way does not remove its scale. It makes that scale easier to divide. The model, worker, rack, storage system, and scheduler become parts of one coherent design that can be measured, illustrated, and improved.



Recommend SemiAnalysis to your readers

Bridging the gap between the world's most important industry, semiconductors, and business.



153 Likes · 2 Restacks

Previous

Discussion about this post

Comments **Restacks**

Write a comment...

© 2026 Dylan Patel · Privacy · Terms · Collection notice
Substack is the home for great culture