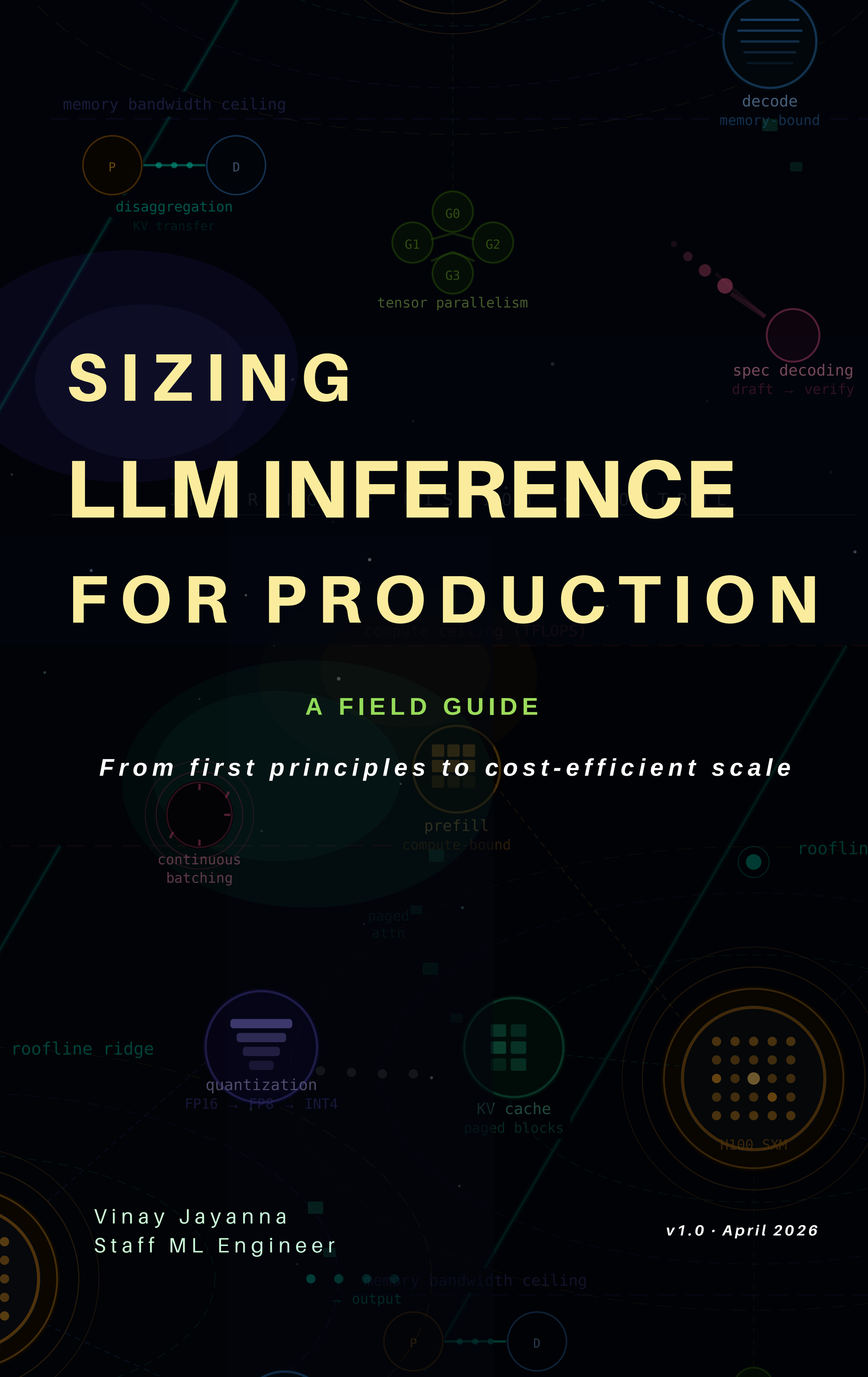
memory bandwidth ceiling
decode
memory-bound
P
D
disaggregation
KV transfer
G0
G1
G2
G3
tensor parallelism
spec decoding
draft → verify
SIZING
LLM INFERENCE
FOR PRODUCTION
A FIELD GUIDE
From first principles to cost-efficient scale
continuous
batching
prefill
compute-bound
roofline ridge
paged
attn
quantization
FP16 → FP8 → INT4
KV cache
paged blocks
H100 SXM
Vinay Jayanna
Staff ML Engineer
v1.0 · April 2026
→ output

WHAT'S INSIDE

Preface

LLM inference has become one of the most consequential infrastructure engineering problems in the industry. The individual techniques are well-documented - quantization, parallelism, batching, KV cache optimization each have deep literature behind them. What is harder to find is a single framework that shows how these pieces fit together, in what order to apply them, and how to turn the output into a GPU count and cost estimate you can stand behind.

This guide is built around that framework. Every decision covered here - memory budgeting, parallelism strategy, quantization format, KV cache configuration - follows from measurable properties of your workload and your hardware. The goal is not to give you configurations to copy. It is to give you the reasoning to derive the right configuration for your specific model, traffic, and constraints - and to understand it well enough to adapt it when things change.

Who this is for. This guide is written for engineers working at the intersection of ML and systems - Staff and Principal ML Engineers, ML Platform Engineers, AI Infrastructure Architects, and Applied Scientists moving into production ownership. It assumes you are comfortable with transformer fundamentals, have hands-on GPU experience, and understand distributed systems concepts like memory hierarchies, throughput, and latency. Prior inference optimization experience is not required - that is what the guide builds from the ground up.

Engineers coming from adjacent areas - distributed systems, cloud infrastructure, or ML platform work - will find the systems reasoning familiar even if the LLM-specific concepts are new. Researchers and applied scientists who want to understand what happens to their models after training - why production behavior differs from evaluation, what drives serving cost, and how architectural choices like Mixture-of-Experts affect deployment economics - will find this a practical bridge between model design and system reality.

How to use this guide. The sections form a decision sequence - each one produces an output that feeds the next. Workload characterization informs memory sizing. Memory sizing constrains parallelism selection. Parallelism selection determines the benchmark configuration. The benchmark produces the operating point. The operating point sizes the fleet. Reading in order builds the full reasoning chain. After that the guide works as a reference - each section on optimization, monitoring, and guardrails stands alone for readers returning to a specific production problem.

A note on the pace of change. This field moves fast - specific numbers, framework defaults, and GPU specs will age. The reasoning framework will not. Read the numbers as illustrations of the method, not as configuration targets to copy verbatim.

Table of Contents

Why Inference Sizing Is Now a Capital Allocation Problem

There is a version of the AI infrastructure conversation that peaked around 2022: which cluster to train on, how many GPUs the training run needs, how to parallelize the backward pass. That conversation was important. It is also largely finished for most organizations.

GPT-4 was trained on an estimated $78-100 million of compute according to the Stanford AI Index Report 2025. That number sounds large. It is a one-time event.

Inference is not a one-time event. Inference runs every second, against every user, for the entire operational lifetime of the model. One analysis projected GPT-4's inference bill at approximately $2.3 billion in 2024 - roughly 15-23× its training cost - simply from the cumulative weight of serving millions of requests per day. Training is a capital expenditure. Inference is an operating expenditure that compounds indefinitely with usage growth.

This shift from training-centric to inference-centric infrastructure spend is now well underway. The inference market is projected to reach $255 billion by 2030. Andreessen Horowitz has documented what they call "LLMflation" - the cost of inference per token has dropped roughly 10× every year since 2022 - yet total inference spend continues to rise because usage is growing faster than prices are falling. Per-token costs are collapsing. Total bills are not.

For organizations running LLMs in production, inference cost is the number that matters most. And yet the discipline of rigorous inference sizing - knowing exactly how many GPUs you need, under what configuration, to serve a defined workload at a defined latency target - remains surprisingly underdeveloped. Most teams either over-provision by instinct ("just add more GPUs") or under-provision until the system breaks under load, then scramble to scale. Neither approach is defensible at production scale.

The "just add more GPUs" instinct is expensive. A fleet running at 40% average GPU utilization is paying roughly 2.5× the cost per token of a well-sized fleet at 70% utilization. One analysis found that chips and staffing together constitute 70-80% of total LLM deployment costs - meaning the hardware sizing decision alone dominates the majority of your production AI budget. Getting it wrong by even 30% has compounding financial consequences over a 3-4 year hardware depreciation cycle.

What makes LLM inference uniquely hard to size is that it is not a single workload. It is two fundamentally different computational workloads - **prefill** and **decode** - running on the same hardware, with conflicting hardware bottlenecks, competing for the same GPU memory, and interacting in ways that are non-obvious without understanding the underlying system dynamics. The GPU memory budget is split between a static component (model weights, loaded once) and a highly dynamic component (KV cache, which grows with every token in every active request). The system that fits in memory at idle will OOM in production if you have not accounted for KV cache at peak concurrency. The system that performs well at batch size 1 will behave completely differently at batch size 64.These are not implementation details. They are the reason inference sizing requires a disciplined framework rather than a rule of thumb - which is what the rest of this guide provides.

1 - Workload Characterization: Know What You're Serving

The expensive sizing mistake in production LLM infrastructure is not a wrong GPU choice or a misconfigured parallelism setting. It is skipping workload characterization entirely and jumping straight to hardware selection. Decisions that follow - memory budget, quantization strategy, parallelism degree, batching configuration - is downstream of understanding what you are actually serving.

Before touching a single configuration parameter, key questions must have concrete, data-driven answers.

What model are you serving?
Parameter count, architecture, and the precision you run it at determine your memory floor before a single request arrives.
On architecture: most models today are either dense or Mixture-of-Experts (MoE). A dense model uses every one of its parameters on every forward pass - Llama-3 70B activates all 70 billion weights every time it generates a token. A MoE model like DeepSeek-V3 works differently: it has 671B total parameters but routes each token through only a small subset of specialized "expert" layers - roughly 37B parameters active per forward pass. The headline size is large but the compute and memory pressure per token is much closer to a 37B model than a 671B one. You need to store all the weights but you only pay compute for the active ones.
On precision: neural network weights can be stored at different levels of numerical accuracy - BF16 and FP16 both use 2 bytes per parameter, FP8 uses 1 byte, INT4 uses half a byte. BF16 has become the standard for training because its wider numerical range prevents instability during backpropagation. For inference, precision choice is purely an engineering tradeoff between memory and quality. The same Llama-3 70B model needs 140 GB at BF16/FP16, 70 GB at FP8, and 35 GB at INT4. Lower precision trades a small amount of output quality for large reductions in memory and faster computation. In production inference, FP8 has become the standard starting point on H100 hardware - it halves memory relative to BF16 with negligible quality impact on most tasks.

What does your input length distribution look like - at P50, P90, and P99? Not the average. The distribution. This matters because KV cache memory is allocated per token per active request. A workload whose P99 input is 16K tokens while its P50 is 512 tokens will behave like two completely different systems under load. The P99 case is what causes OOM failures in production.

What does your output length distribution look like? Output length determines how long each request occupies a decode slot, which directly governs throughput. A system serving 50-token outputs can handle dramatically more concurrent requests than one serving 2,000-token outputs on identical hardware.

What is your peak requests per second? Not your average. See section 1.4 on why this distinction is sharper than most teams expect.

What are your latency SLOs, and which metric matters most? TTFT (time to first token), ITL (inter-token latency), and E2E (end-to-end) latency are three different things driven by three different system properties. A system optimized for TTFT may actively degrade ITL, and vice versa. Conflating them leads to configurations that satisfy benchmarks but fail users.

Online streaming or offline batch? These are not different points on the same spectrum - they are fundamentally different optimization targets that call for different architectures entirely.

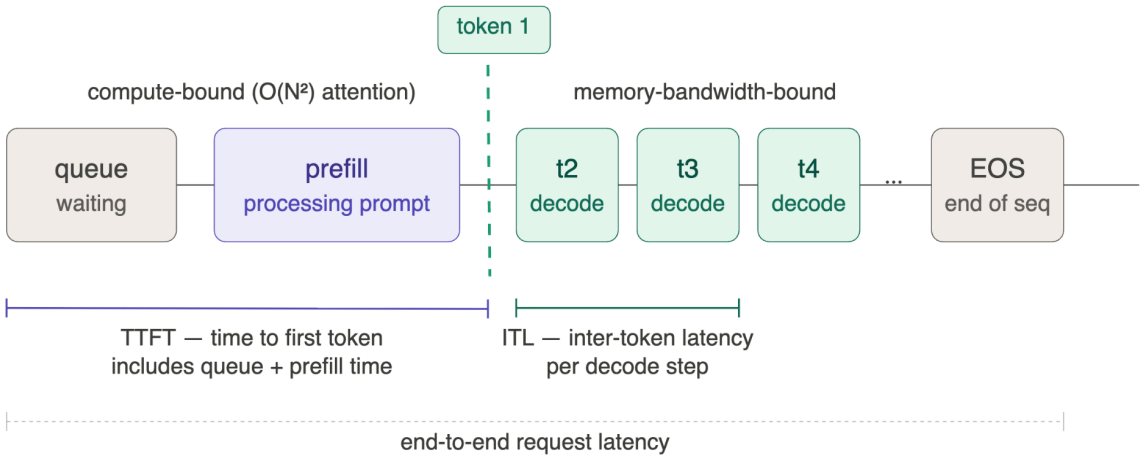

Figure 1.1 - Request lifecycle: TTFT, ITL, and end-to-end latency
A single request passes through three distinct phases. It first waits in queue - this time is often invisible in benchmarks but fully visible to users. Prefill then processes the entire prompt in one compute-bound pass (O(N²) attention), producing the first token; the time from request arrival to this first token is TTFT - time to first token. Each subsequent token is generated one at a time in the decode phase, which is memory-bandwidth-bound rather than compute-bound because the full model weights must be loaded from HBM on every step; the time between consecutive tokens is ITL - inter-token latency. End-to-end latency spans the entire journey from queue entry to the final EOS token. TTFT, ITL, and E2E are three distinct metrics driven by three different system properties - optimizing one can actively degrade another.

1.1 Why P99 Input Length Matters More Than P50

P50 input length is useful for capacity planning at steady state. It is useless for preventing production failures.

Consider a serving system sized for a P50 input of 512 tokens. Under normal load it runs comfortably. Then a burst of P99 requests arrives - each with 32K tokens of context, perhaps from a RAG pipeline retrieving a full document set. Each of those requests allocates 32K × KV-cache-bytes-per-token of GPU memory. If 20 such requests land simultaneously on an instance sized for typical traffic, KV cache is exhausted, new requests are rejected or preempted, and the failure propagates across the batch. The system did not fail because the hardware was wrong. It failed because the sizing assumption was wrong.

The correct approach is to size KV cache headroom for P95 or P99 input length at your target peak concurrency. The median tells you what your system handles most of the time. The tail tells you what breaks it. In production LLM deployments, OOM failures almost always originate at the tail - a workload characteristic that is invisible if you only measure averages. Every sizing decision in this guide is built around that principle: design for the distribution, not the mean.

1.2 Four Workload Archetypes: Each With a Different Hardware Bottleneck

Not all LLM workloads stress the system the same way. There are four distinct archetypes, each with a different dominant bottleneck and a different sizing profile.

Chat and conversational AI - short inputs, short outputs, highly latency-sensitive. TTFT dominates the user experience because humans perceive the pause before the first word. ITL is largely invisible because modern inference engines generate tokens faster than humans read. The hardware bottleneck is prefill throughput at low concurrency, and the key metric is TTFT under load.

RAG pipelines - long inputs (retrieved documents plus query, often filling most of the context window), short to medium outputs. The dominant cost is prefill: processing thousands of input tokens to produce a few hundred output tokens. This is compute-intensive and benefits most from high TFLOPS and quantization that reduces weight movement during prefill. KV cache pressure is high per request but duration is short.

Code generation and long-form reasoning - short to medium inputs, long outputs. The dominant cost is decode: generating hundreds or thousands of tokens autoregressively. This is memory-bandwidth-bound - the system spends most of its time moving model weights from high-bandwidth-memory (HBM) to compute units for each decode step. The key metric is ITL and sustainable decode throughput at high concurrency. This archetype benefits most from high HBM bandwidth GPUs and large batch sizes that get more work done per weight-loading trip.

Agentic and multi-step workflows - the archetype that breaks every simple sizing model. A single user action triggers a chain of sequential LLM calls: planning, tool invocation, result interpretation, re-planning. Context accumulates across turns. Total token consumption per user session is an order of magnitude higher than a single-turn interaction. The challenge here is not peak RPS in the traditional sense - it is sustained GPU occupancy across a session with variable-length, dependent calls. Sizing for agentic workloads requires modeling the entire session token budget, not just individual request characteristics.

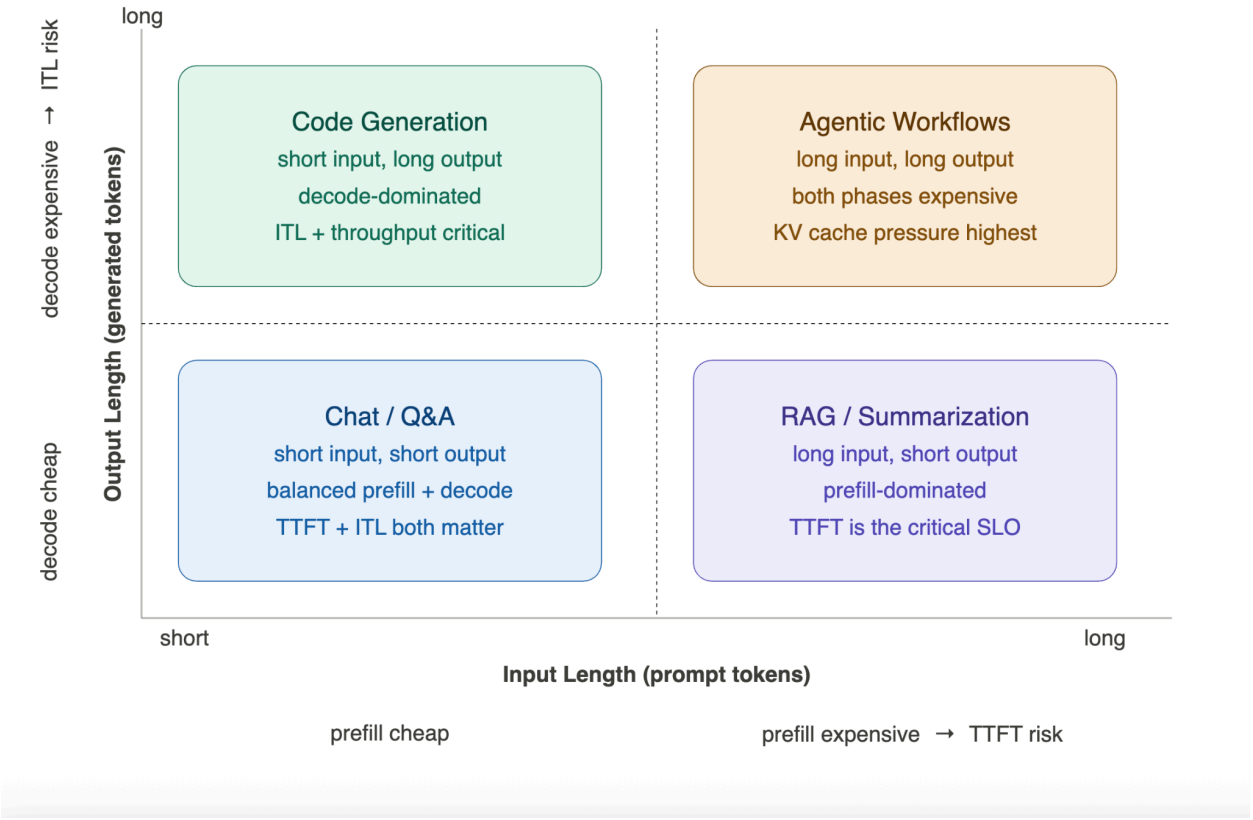

Figure 1.2 - Four workload archetypes: input/output length determines your bottleneck
Position on this grid tells you which hardware resource will constrain you first. Moving right increases prefill cost - longer prompts mean more compute and higher TTFT risk. Moving up increases decode cost - longer outputs mean more HBM bandwidth pressure and ITL risk. Chat/Q&A sits at the origin: balanced and forgiving, both phases cheap. RAG/Summarization moves right: prefill-dominated, TTFT is the SLO that breaks first. Code Generation moves up: decode-dominated, ITL and sustained throughput are what matter. Agentic Workflows occupy the top-right corner: both phases expensive, KV cache pressure highest, and the standard per-request sizing model breaks down entirely.

1.3 Sizing for Peak, Not Average: Request Arrival Patterns

Production traffic is bursty. The Poisson distribution is a reasonable approximation for arrival patterns in many systems, but it has a key property that sizing teams frequently ignore: the gap between average arrival rate and peak arrival rate is largest at low average rates.

A representative enterprise deployment illustrates how badly this plays out in practice. At an average of 65 requests per second, the P95 peak is 160 RPS - roughly 2.5× the mean, and P99 reaches 180 RPS. The overnight trough pulls the daily average down significantly while the business hours peaks drive the actual capacity requirement. Sizing for the average means your fleet is undersized for approximately 8 hours every working day.

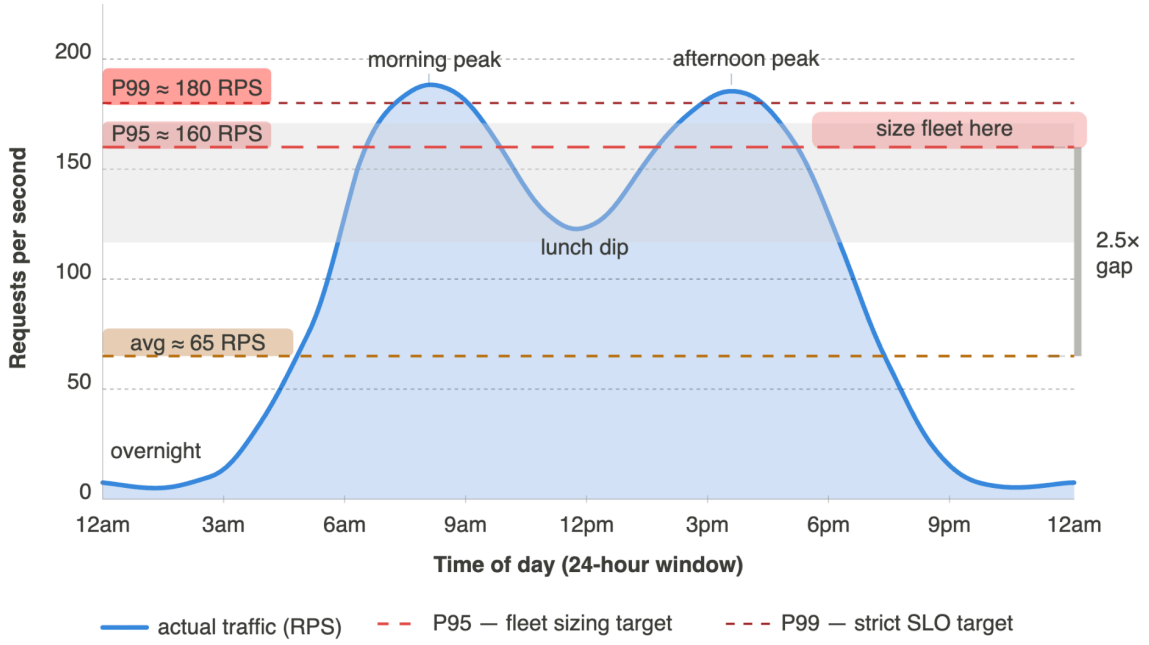

Figure 1.3 - Real enterprise traffic over 24 hours. *The gap between average RPS (orange) and P95 RPS (red) is the zone where sizing for average guarantees SLO violations. Both peaks - morning and afternoon - exceed P95 while the overnight trough pulls the average down significantly.*

The practical implication: gather actual traffic data, compute the P95 or P99 peak, and size GPU capacity to sustain that peak without SLO degradation. Every downstream calculation in this guide - memory budget, parallelism degree, batch size - assumes P95 as the input, not the average. Sizing for average is not a conservative choice; it is a scheduled outage.

1.4 Online Streaming vs. Offline Batch: Two Optimization Targets

Online streaming and offline batch are not two points on the same dial - they are different optimization problems that require different architectures, different operating points, and different success metrics.

In online streaming mode - where responses stream token by token to a waiting user - generation typically happens faster than human reading speed. Fast readers consume roughly one token every 90ms. Production inference engines generate a token every 15-25ms per request under moderate load. This means ITL is largely invisible to users in streaming mode: the bottleneck on user experience is TTFT, the delay before the first token appears. Systems optimized for online streaming should minimize TTFT, accept some throughput sacrifice to do so, and favor lower concurrency operating points with more aggressive prefill prioritization.

In offline batch mode - document processing, dataset annotation, bulk summarization - no user is waiting. TTFT is irrelevant. The optimization target is pure throughput: maximize tokens generated per GPU-hour, minimize cost per million tokens. Systems optimized for offline batch should operate at maximum sustainable concurrency, use the largest batch

sizes that fit in memory, and accept higher latency per request in exchange for higher aggregate throughput.

The failure mode in production is running both workload types on a shared fleet - batch jobs saturate concurrency, TTFT spikes, and latency SLOs degrade while the GPU sits at high utilization doing the wrong work. Offline batch should either fill idle capacity during off-peak hours on latency-optimized fleets, or route to a dedicated throughput fleet entirely.

1.5 Model Routing and Cascading: When One Model Is Wrong

Every sizing calculation in this guide assumes a single model handling all requests - and for many deployments, that assumption holds. Before accepting it for yours, ask one question: **do all your requests actually need the same model?**

In most production workloads, request complexity is highly skewed. A customer service chatbot handles thousands of "*what are your opening hours?*" queries for every one complex multi-step reasoning request. A coding assistant handles hundreds of simple autocomplete requests for every architectural design question. If you size a fleet of 70B-parameter GPUs to handle all of these uniformly, you are spending frontier model compute on requests that a 7B model could handle equally well - at roughly 10× lower cost per token.

The fix is a routing layer in front of your fleet that classifies each incoming request by complexity and directs it to the appropriate model tier. This pattern goes by two names that are related but not interchangeable.
Model routing makes the tier decision upfront - the router classifies the request, picks a tier, and sends it there directly.
Model cascading attempts the small model first and escalates to the large model only when the response falls below a quality threshold.

Routing is lower latency because the decision happens once before inference begins. Cascading is more adaptive because it uses actual output quality as the signal, but it pays for that adaptability with a second inference call on every escalated request - which can be 20-40% of traffic depending on your quality threshold. Most production deployments use routing rather than cascading for latency-sensitive workloads. Cascading is better suited to offline batch pipelines where the escalation cost is acceptable and output quality is the primary constraint.

How it works in practice:

Step 1 - Define complexity tiers. Most deployments need only two: a small fast model (7B-13B, heavily quantized, commodity GPUs) for straightforward requests, and a large capable model (70B+, H100s) for complex ones. Some add a third middle tier for moderate complexity.

Step 2 - Build or use a router. The router is a lightweight classifier - either a small fine-tuned model, a rules-based system (input length thresholds, keyword matching), or a learned routing model like RouteLLM - that assigns each request to a tier before it enters the serving stack. The router's latency budget is typically 5-20ms; anything more starts eating into TTFT.

Step 3 - Size each tier independently. This is the critical implication for this guide. Once you introduce routing, you are no longer sizing one fleet - you are sizing two (or more) independent fleets, each with its own memory floor, benchmark curve, SLO-constrained operating point, and GPU count. The sizing algorithm in Section 9 runs separately for each tier.

Cascading follows the same tier structure but reverses the decision logic. Instead of classifying upfront, every request enters the small model tier first. A quality evaluator - typically a lightweight scorer or a confidence threshold on the output - decides whether the response is acceptable. If not, the request escalates to the large model tier and inference runs again from scratch. The sizing implication is different from routing: your small model tier must be sized to handle 100% of peak RPS since every request passes through it, while the large model tier is sized only for the escalation fraction. Cascading is rarely worth the complexity for online workloads but may become attractive for offline batch pipelines where output quality is the primary constraint and the escalation latency penalty is irrelevant.

The TCO impact is substantial. If 70% of traffic routes to a 7B tier running on commodity GPUs and 30% routes to 70B on H100s, your blended cost per token is roughly `0.7×(small cost) + 0.3×(large cost)` - a significant reduction versus uniform large-model serving. RouteLLM's benchmarks report 40-85% cost reduction in practice depending on quality threshold, which is consistent with this math.

What routing does not fix. Routing adds a new failure mode: misclassification. A complex request routed to the small model produces a poor response; a simple request routed to the large model wastes compute. The router's quality threshold is a dial between cost savings and quality risk - set it too aggressively and you degrade user experience; set it too conservatively and you capture little of the potential savings. Always measure small-model response quality on your specific workload before committing to a routing threshold.

Routing also adds operational complexity - two or more fleets to deploy, monitor, scale, and maintain instead of one. For small deployments this overhead is not worth it. The crossover point is roughly when your large-model fleet would otherwise exceed 8-10 GPUs and your traffic has a measurable skew toward simpler requests.

When to consider routing:

Signal	Recommendation
Traffic is highly homogeneous in complexity	Skip routing - overhead not justified
Clear bimodal complexity distribution (many simple, few complex)	Strong routing candidate
Large model fleet exceeds 8 GPUs	TCO savings likely justify routing complexity
Response quality on small model is measurable and acceptable	Proceed with routing

Latency budget is extremely tight (< 200ms TTFT)	Add router latency to your TTFT budget model before committing; rule out routing if the router alone consumes > 10% of budget.

2 - GPU Memory Sizing: The Hard Constraint

GPU VRAM is not a soft limit you can negotiate around with clever configuration. It is a binary constraint. The system either fits or it does not start. No batching strategy, no parallelism trick, no serving framework feature helps you if the total memory footprint exceeds available VRAM. This makes memory sizing the first engineering calculation, not the last.

When an LLM inference engine starts up, it must allocate four distinct categories of GPU memory before it can serve a single request. These four categories have completely different characteristics - some are fixed at load time, some grow dynamically with traffic, some are model-determined, some are workload-determined - and conflating them is the root cause of most production memory miscalculations.

1. **Model weights** are the parameters loaded from disk. This is the number everyone calculates. It is fixed the moment the model loads and does not change regardless of traffic. It is also the easiest component to reason about because it depends only on parameter count and numerical precision.

2. **KV cache** is the stored key and value tensors from the attention mechanism, maintained for every token in every active request. Unlike weights, KV cache is not a property of the model - it does not appear on model cards and cannot be measured offline. It grows dynamically at runtime as a function of sequence length, concurrency, and the number of attention layers. At high concurrency with long context, KV cache routinely exceeds model weight memory by a significant margin. This is the component that will exhaust your VRAM budget before anything else does.

3. Activations are the intermediate tensors computed during the forward pass - the layer inputs, outputs, and attention matrices that exist transiently as each token is processed. They are not stored across requests like KV cache; they exist only during the computation of a single forward pass. However at high batch sizes they accumulate across all layers simultaneously, and their memory footprint scales with batch size, sequence length, and model width. At large batch sizes they become non-trivial - typically 15-30 GB for a 70B model at production concurrency - and ignoring them produces a memory budget that is systematically optimistic.

4. Framework overhead is everything the serving runtime consumes that has nothing to do with the model or the requests. This includes the CUDA runtime itself, the driver context, internal memory pools and allocators maintained by the serving framework - vLLM, TensorRT-LLM, Triton - plus pre-allocated buffers for async scheduling, and the memory fragmentation that accumulates over time in long-running processes. This component is opaque, vendor-specific, and almost never documented. In practice it runs 10-20 GB for

mature frameworks on H100s and should be treated as a fixed tax on every deployment regardless of model size.

These four components sum to your total VRAM requirement:

```
Total VRAM = Model Weights + KV Cache + Activations + Framework Overhead
```

The formula is simple. The difficulty is that three of the four terms are functions of runtime variables - concurrency, sequence length, batch size - not static model properties. You cannot calculate this from a model card. You must calculate it from your workload characterization.

To make this concrete, consider Llama-3 70B - one of the most commonly deployed frontier-class models - at 50 concurrent requests with 8K context. The numbers are representative of what a real production sizing exercise produces. Here is what the four components actually produce:

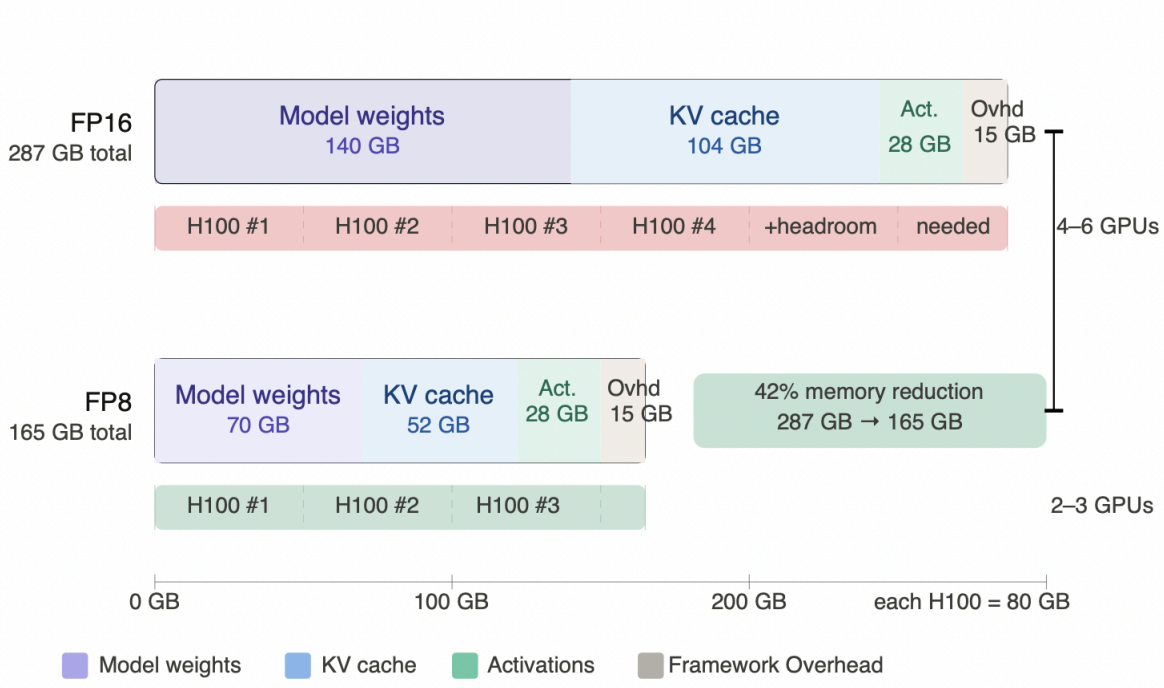

Figure 2.1 - GPU memory breakdown: FP16 vs FP8 for Llama-3 70B (50 concurrent requests, 8K context) *Quantization from FP16 to FP8 halves both model weights and KV cache, reducing total memory from 287 GB to 165 GB. Activations and framework overhead are unaffected by weight precision. The GPU count drops from 4-6 H100s to 2-3 - the single highest-leverage memory decision available before touching throughput.*

The sections that follow examine each component in detail - the formula that governs it, the runtime variables that affect it, and the mistakes teams make when they underestimate it.

2.1 Model Weights: The Floor

This is the one everyone calculates. Model weight memory is determined by parameter count and numerical precision.

```
Model memory (bytes) = num_parameters × bytes_per_dtype
```

Precision	Bytes per parameter
FP32	4
FP16 / BF16	2
INT8	1
INT4	0.5

In practice: Llama-3 8B in FP16 needs 16 GB. Llama-3 70B in FP16 needs 140 GB. Llama-3 70B in INT4 needs 35 GB - the difference between requiring a full DGX node and fitting on a single GPU with room to spare. Quantization's impact on memory is this stark, which is why Section 4 covers it as a first-class sizing lever rather than a post-hoc optimization.

The formula is correct but incomplete. Five properties of model weight memory that production engineers learn the hard way:

The alignment tax is real. GPU memory allocators cannot pack tensors like books on a shelf - each tensor requires padding to align to memory boundaries, and that padding accumulates silently across hundreds of layers. In practice, resident weight memory runs 3-8% above what the formula predicts. On a 140 GB model that is 4-11 GB of memory you never explicitly allocated. On a tight configuration, that invisible overhead is what tips you into an OOM at load time.

Weight memory is a loading event, not just a budget line. Most engineers think about weight memory as a static number - the model either fits or it doesn't. But that memory has to travel from storage to GPU before the first request can be served, and the path matters as much as the size.

Storage path	Effective bandwidth	70B FP8 (70 GB) cold start	70B FP16 (140 GB) cold start
Local NVMe (PCIe Gen4)	~12.5 GB/s	~6 seconds	~11 seconds
Network-attached (10 GbE)	~1.1 GB/s	~64 seconds	~2 minutes
Object storage (S3/GCS) with container init	~0.5-1.0 GB/s effective	2-5 minutes	4-10 minutes

The object storage row reflects real Kubernetes conditions - raw transfer bandwidth is only part of the cost. HTTP chunked transfer, metadata operations, framework initialization, and container startup all stack on top, pushing cold start times to 2-10 minutes for 70B-class models. This is orders of magnitude longer than web service cold starts, and it breaks

reactive autoscaling entirely: by the time a new instance is ready, the traffic spike has already peaked and your SLO has already been violated.

Scale-from-zero is almost never the right pattern for LLM fleets for this reason. Unlike stateless web services where scale-from-zero is economically attractive, LLM fleets pay a 2-10 minute cold start penalty on every scale-up event from zero - a penalty that makes it operationally untenable for any workload with predictable traffic.

Quantization and local NVMe caching of model artifacts are therefore not just cost and memory optimizations - they are autoscaling responsiveness decisions. A 35 GB FP4 model loads in under 3 seconds from local NVMe. A 140 GB FP16 model pulling from object storage may take 10 minutes. That gap determines whether your fleet can absorb a traffic spike before the SLO is breached. Section 9 covers the production patterns - predictive scaling, minimum warm instance count, and NVMe artifact pre-caching - that manage this constraint operationally.

Precision mismatches create transient spikes. Think of it like moving furniture through a doorway - you need more space during the move than you need once everything is in place. Loading FP16 weights and casting to BF16 at runtime briefly holds both representations in memory simultaneously. The same happens when certain layers are kept in higher precision while the rest serve in INT8. The spike is short-lived but it happens at the worst possible moment - initialization - when you have no requests yet to justify the allocation. A configuration that fits at steady state can OOM before it ever serves a token.

LoRA fundamentally changes the multi-tenant weight equation. Base weights are loaded once and shared across every adapter variant running on that instance. Each LoRA adapter adds only its delta - typically 1-5% of base model size. The practical implication is that serving ten fine-tuned variants of Llama-3 70B costs approximately the same weight memory as serving one. This is one of the most underutilized cost levers in multi-tenant deployments. Subsequent sections cover the full model, but the property originates here.

Serialization format affects load behavior, not resident size. Once loaded, a model in safetensors and the same model in a pickle-based format occupy identical GPU memory. The difference is how they get there. Safetensors supports memory-mapped loading - the OS pages in only what is needed, and the process never holds a full second copy during deserialization. Pickle-based formats require a full deserialization pass that temporarily allocates a second buffer alongside the model being loaded. On a memory-constrained system, that transient second allocation is enough to fail initialization even when the model itself fits.

Overall model-weight memory is the floor. The system cannot initialize below this threshold. Everything that follows is overhead stacked on top of it - and the next component will, in most production workloads, exceed it.

2.2 KV Cache: Where Production Systems Run Out of Memory

This is the component most teams underestimate, often catastrophically. KV cache does not appear in model cards. It is not a static property of the model. It grows dynamically at

runtime with every token in every active request - and at high concurrency with long context, it routinely exceeds model weight memory by a significant margin.

The formula:

$$\text{KV\_cache (bytes)} = 2 \times \text{num\_layers} \times \text{num\_KV\_heads} \times \text{head\_dim} \times \text{seq\_len} \times \text{concurrency} \times \text{bytes\_per\_element}$$

The factor of 2 is for both the key and value vectors stored per layer. Everything else scales with your workload.

A concrete example grounds the math. Llama-3 70B in FP16 has 80 layers, 8 KV heads, and a head dimension of 128 - working out to approximately 0.26 MB of KV cache per token per request.

At a typical production load of 50 concurrent requests with 8K-token inputs, that is 106 GB of KV cache sitting alongside the 140 GB of model weights on the same system. Already more than 70% of the model's own footprint, just from active requests.

Now extend the context. At 32K tokens the KV cache grows to 416 GB at the same concurrency - three times the model weights. At 128K context, just 4 concurrent requests push KV cache past model weights entirely. The diagram below makes this viscerally clear: short-context workloads stay manageable, but as context length grows the KV cache stops being overhead and becomes the dominant memory consumer in the system.

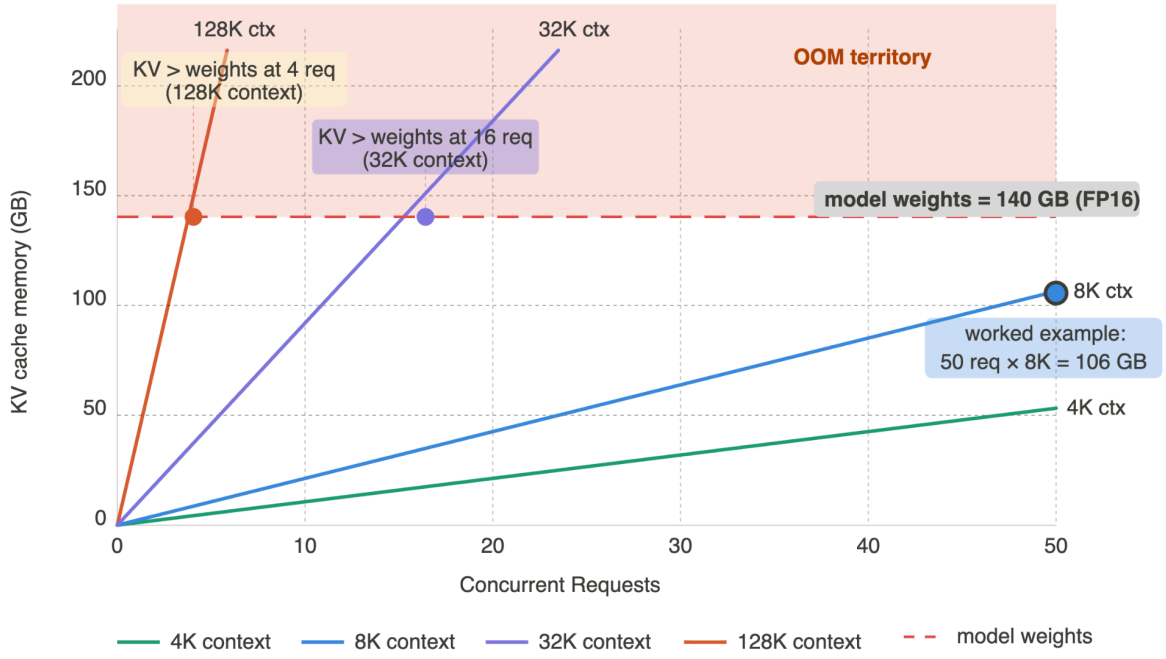

Figure 2.2 - KV cache memory vs concurrency for Llama-3 70B (FP16)
Each line represents a different context length. KV cache grows linearly with concurrency - but the slope scales directly with context length, making long-context workloads disproportionately memory-intensive. At 128K context, model weights are exceeded at just 4 concurrent requests. At

32K context, the crossover happens at 16. Only short-context workloads (4K-8K) stay below model weight memory at typical production concurrency. Size for your P95 context length, not your average.

This is not an edge case. It is the normal operating condition for any RAG pipeline or long-context deployment running at meaningful concurrency. The system sized for the model alone will OOM under realistic load. Every production deployment must calculate both components and sum them before selecting a GPU configuration.

One architectural variable has an outsized effect on KV cache size that is easy to miss if you only read the model card: the ratio of query heads to KV heads. During decode, each token attends to all previous tokens to decide what information to carry forward. Rather than doing this once, transformer models do it multiple times in parallel - each "head" captures a different type of relationship (syntactic, semantic, positional). Each head maintains its own Key and Value vectors, which are what get stored in the KV cache. With 64 heads and a 4K-token context, you are storing 64 sets of K and V vectors per token per layer - which is where KV cache memory comes from.

Multi-Head Attention (MHA) is the standard: every query head has its own dedicated KV head. A model with 64 attention heads stores 64 KV head pairs per token. During decode, every one of those must be loaded from HBM on every step.

Grouped Query Attention (GQA) makes a practical observation: multiple query heads can share the same Key and Value vectors without meaningfully hurting quality - they still ask different questions (each has its own Q vector) but consult the same shared reference material (shared K and V). Think of a committee where every member forms their own opinion independently, but they all read from the same shared briefing document rather than maintaining separate personal copies. GQA reduces KV head count dramatically while preserving model quality.

Llama-3 70B has 64 query heads but only 8 KV heads - an 8× reduction in KV cache size relative to a standard MHA model of the same scale. Modern architectures including Llama-3, Mistral, Qwen, and Gemma all use GQA for exactly this reason - it is one of the primary reasons these models are more memory-efficient during serving than older architectures of comparable parameter count.

When estimating KV cache memory for a model you have not deployed before, always check the KV head count explicitly - not the total attention head count. They are often not the same number, and using the wrong one will cause you to overestimate KV cache memory significantly.

2.3 Activations: The Necessary but Manageable Component

Every forward pass through a transformer layer produces intermediate tensors that must stay resident in memory until the next operation consumes them. These are not stored across requests the way KV cache is - they exist only for the duration of a single forward pass - but at large batch sizes they accumulate across all layers simultaneously and their footprint is non-trivial.

The historical problem was severe. In standard attention, every token must attend to every other token to decide what information to carry forward. With a sequence of N tokens, that produces an N×N attention matrix - one entry for every token-to-token relationship. At 8K tokens that matrix is 64 million entries. At 32K tokens it is over a billion. Memory consumption scaled as O(seq²), meaning doubling context length quadrupled activation memory. Long-context workloads were genuinely dangerous to run at scale under the original formulation.

FlashAttention eliminates the quadratic scaling, but it is worth understanding exactly how. The insight is that you never actually need the full N×N matrix in memory at once - you need the *result* of multiplying it by the value vectors, which is a much smaller tensor. FlashAttention computes this result incrementally by processing the attention computation in small blocks that fit entirely in the GPU's fast on-chip SRAM, which is orders of magnitude faster to access than HBM but also much smaller. Each block computes its contribution to the final output, accumulates the result, then discards the intermediate attention scores before moving to the next block. The full N×N matrix is never materialized anywhere. Think of it like computing a sum of a million numbers by adding them in groups of a hundred - you never need to write down all million numbers simultaneously, just maintain a running total.

The memory consequence is that activation scaling drops from O(seq²) to approximately O(seq) - linear rather than quadratic. Doubling context length roughly doubles activation memory rather than quadrupling it. This is what makes 32K and 128K context workloads operationally viable on real hardware.

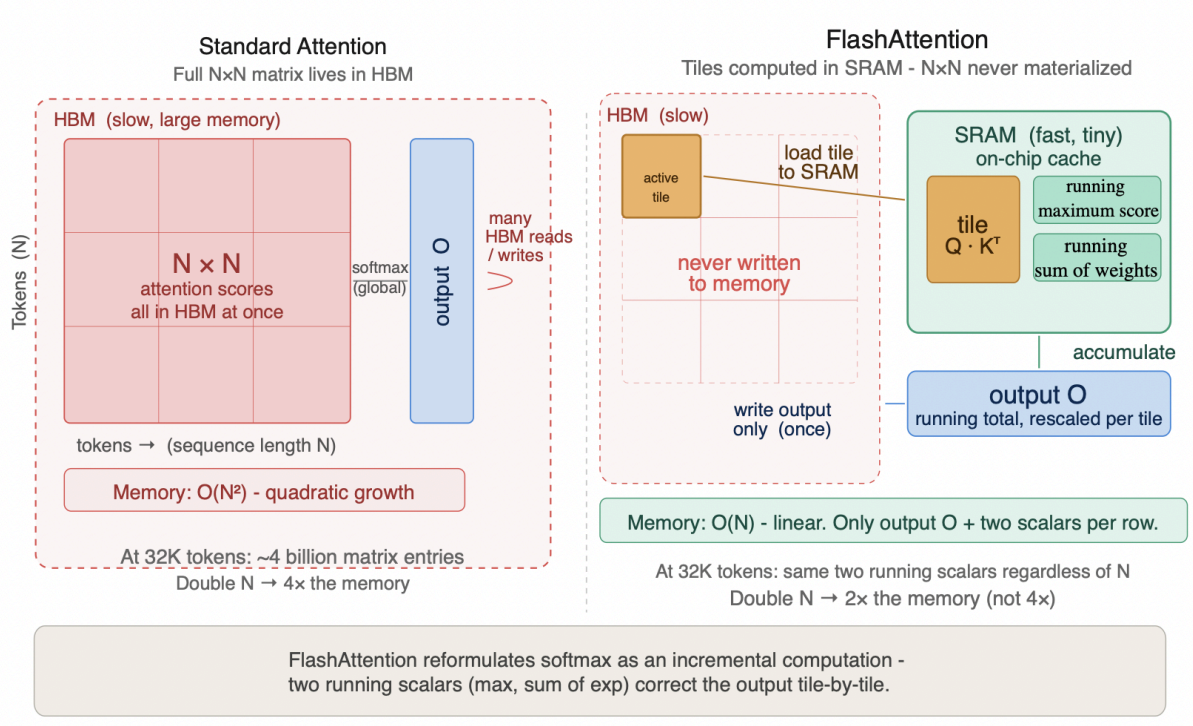

Figure 2.3 - Standard attention vs FlashAttention memory model
Standard attention materializes the full N×N score matrix in HBM before softmax can run - memory grows quadratically with sequence length. At 32K tokens that is roughly 4 billion matrix entries.

FlashAttention eliminates this by computing attention in tiles that fit entirely in on-chip SRAM, maintaining only two running scalars per row - a running maximum and a running sum of exponentials - that allow softmax to be computed correctly without ever seeing the full matrix. Memory drops from $O(N^2)$ to $O(N)$. The N×N matrix is never written anywhere. This is what makes 32K and 128K context workloads physically viable on current hardware.

FlashAttention is now standard across all major inference frameworks - vLLM, TensorRT-LLM, and Triton all use it by default. You do not need to enable it manually. But understanding what it does matters for sizing because it changes the activation memory model entirely: you are no longer dealing with a component that explodes with context length, you are dealing with one that grows predictably and stays bounded relative to model size.

In practice, budget approximately 20% of model weight memory for activations under FlashAttention. For Llama-3 70B that is roughly 28 GB - significant, but predictable and stable across context lengths in a way that KV cache is not. Unlike KV cache, activations do not creep upward as concurrency or context grows beyond your sizing assumptions. Once you have calculated model weight memory, the activation budget follows directly from it.

This makes activations the most manageable of the four components: not negligible, not ignorable, but calculable from a single number you already know.

2.4 Framework Overhead: The Remainder You Cannot Ignore

Every inference framework - vLLM, SGLang, TensorRT-LLM - is not just a thin wrapper around model execution. It is a runtime system that manages memory pools, request queues, block tables, batching state, and CUDA contexts. All of that infrastructure lives in GPU memory.

Think of it like moving into a new apartment. Before you bring a single piece of furniture, the building has already consumed square footage for the lobby, the stairwell, the electrical room, and the maintenance closet. You never get to use that space for living. Framework overhead works the same way - it is the infrastructure tax paid before the workload begins, and it is non-negotiable.

What drives the number varies by framework. vLLM pre-allocates block tables for PagedAttention proportional to your configured memory pool size. TensorRT-LLM pre-allocates engine execution buffers at build time, sized to your declared maximum batch size and sequence length - build for batch 128 at 32K context and those buffers are reserved regardless of actual traffic. SGLang maintains a radix cache structure for prefix reuse that adds its own allocation on top of the base CUDA runtime.

The practical budget is 5-10% of total VRAM on top of the other three components. On an 80 GB H100 that is 4-8 GB that is structurally unavailable regardless of what the model card says. This is why systems that look viable on paper occasionally fail to initialize - the calculation accounted for weights, KV cache, and activations, but not the framework claiming its share first.

2.5 Multi-LoRA Serving: Memory Implications

For deployments serving a single model variant, the four components above complete the memory budget. Most enterprise deployments do not serve a single variant. A customer service platform might run dozens of domain-specific fine-tunes of the same base model - one for billing, one for technical support, one per language. Fine-tuning a 70B base model for each use case and deploying separate instances is prohibitively expensive. Fine-tuning does not require retraining the full model - it produces a small set of weight adjustments called an **adapter** that captures only the task-specific changes, typically 50-200 MB versus 140 GB for the full 70B weights. Multi-LoRA serving solves this by sharing the base model weights across all tenants while loading only the lightweight adapter weights per request - a single GPU fleet serves dozens of fine-tuned variants without duplicating the base model.

The memory math is straightforward but frequently miscalculated in production:
Base model memory - unchanged. The full model weights at target precision load once and are shared across all LoRA variants.

LoRA adapter memory - each adapter adds a small memory overhead. A LoRA adapter for a 70B model with rank r=16 typically adds 50-200 MB per adapter depending on which weight matrices are adapted (Q, K, V, FFN, or all).

The rank (r) controls the adapter's expressive capacity. Each weight matrix in the model is a large 2D grid of numbers - LoRA does not modify it directly. Instead it learns two small rectangular matrices that together approximate only the *change* needed from fine-tuning: one narrow matrix (rank × full dimension) and one flat matrix (full dimension × rank). Multiplied together they reconstruct a full-size update, but constrained to patterns expressible within r independent directions. Think of it like describing a complex piece of music using only r instrument tracks - rank=4 gives you 4 tracks to approximate the full arrangement, rank=64 gives you 64. More tracks means finer detail but larger memory footprint. The base model weights never change - you are only storing and swapping the lightweight adapter matrices.

Most production deployments use r=8 to r=64. Crucially, only the adapters for currently active requests need to reside in GPU memory - inactive adapters can be paged out to CPU memory and loaded on demand.

Peak adapter memory - the binding constraint is not the total number of adapters but the maximum number of distinct adapters simultaneously active in the batch. If your batch of 50 concurrent requests spans 8 distinct LoRA variants, you need memory for 8 adapters simultaneously, not 50.

```
Total memory = base_model_memory + (max_simultaneous_adapters ×
               adapter_size) + KV_cache + activations
```

The failure mode teams hit in production. A deployment serving 100 LoRA adapters assumes adapter memory is negligible because each adapter is small. Under bursty traffic, requests for many distinct adapters arrive simultaneously, all 100 adapters get loaded into GPU memory concurrently, and the deployment OOMs - not because of the base model or KV cache, but because of adapter accumulation. The fix is to cap

`max_simultaneous_adapters` in your serving configuration and implement LRU eviction for inactive adapters

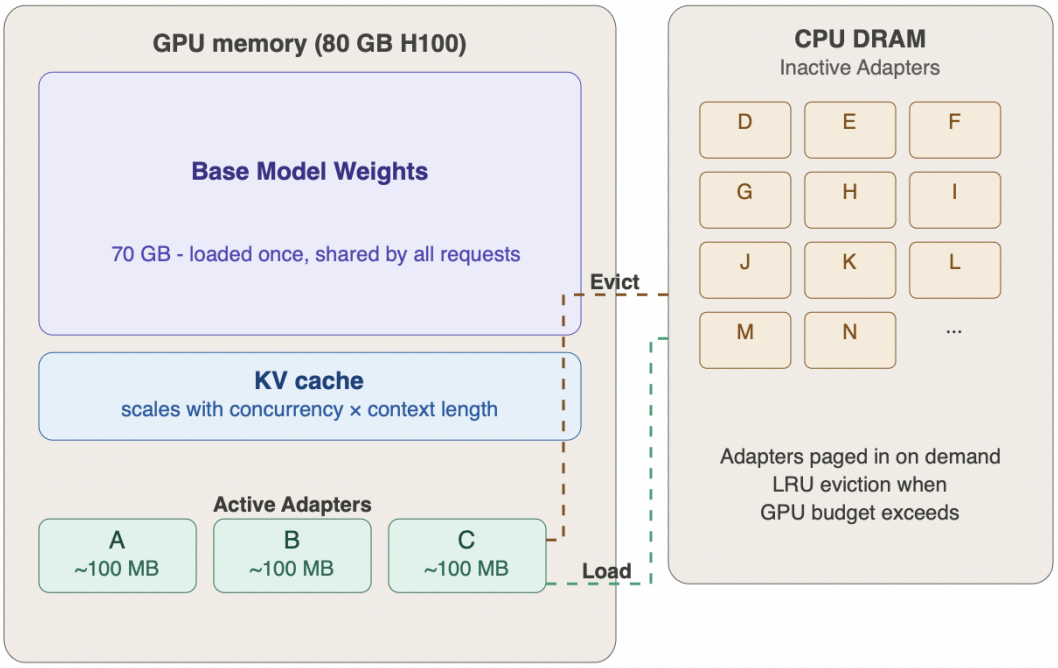

Figure 2.5 - Multi-LoRA serving memory architecture
The base model loads once and is shared across all tenants. Only adapters for currently active requests reside in GPU memory - inactive adapters are paged to CPU DRAM and loaded on demand via LRU eviction. The binding constraint is not the total number of adapters but the maximum number simultaneously active in the batch. The failure mode: bursty traffic causes all adapters to load concurrently, exhausting GPU memory - not from the base model or KV cache, but from adapter accumulation.

Framework support. vLLM supports multi-LoRA serving natively with configurable `max_loras` (number of adapters in GPU memory simultaneously) and `max_lora_rank`. SGLang supports multi-LoRA with similar controls. Both implement adapter paging - loading adapters from CPU to GPU on demand and evicting LRU adapters when the GPU memory budget is exhausted.

2.6 Putting It Together: The Minimum GPU Count Formula

Every component in this section was building toward a single calculation. You now have everything you need.

$$\text{Min GPUs} = \text{ceil}((\text{model\_weights} + \text{KV\_cache\_at\_peak\_concurrency} + \text{activations} + \text{overhead} + (\text{max\_simultaneous\_adapters} \times \text{adapter\_size})) / \text{GPU\_VRAM})$$

The KV cache term is the variable that changes everything. It is a function of your workload - context length, concurrency, and model architecture - not a property of the model alone. Two teams running the same model on the same GPU will get completely different minimum GPU counts if their concurrency and context length assumptions differ. This is why workload characterization in Section 1 comes before GPU selection, not after.

Worked example: Llama-3 70B in FP16, 50 concurrent requests at P95 input length of 8K tokens, H100 80 GB.

Model weights come in at 140 GB. KV cache at this concurrency and context length works out to `50 × 8,000 × 0.26 MB ≈ 104 GB`. Activations add roughly 28 GB and framework overhead another 15 GB. Total: approximately 287 GB - which requires `ceil(287 / 80) = 4 H100s` at minimum, with essentially no headroom for anything.

If you are serving 8 simultaneous LoRA adapters at rank=16, add roughly 0.8 GB - negligible at this scale, but meaningful at high rank or large simultaneous adapter counts.

Now apply FP8 quantization. Model weights drop to 70 GB, KV cache drops to 52 GB, total falls to approximately 165 GB, and 2-3 GPUs become viable. That is the leverage quantization provides at the memory sizing stage - before throughput, before latency, before any other optimization. It is a GPU count decision, not a quality tuning decision.

On headroom. The minimum GPU count is not your deployment GPU count. A system running at 95% VRAM utilization is one traffic burst away from an OOM cascade. Add 30-50% headroom beyond the calculated minimum: enough to absorb spikes above P95, keep the system away from the memory cliff edge, and survive framework version updates that quietly increase overhead between deployments. For the FP16 example above, that headroom recommendation moves the practical minimum from 4 GPUs to 6. For the FP8 case it moves from 2-3 to 3-4 - still a meaningful saving over the FP16 baseline.

3 - The Roofline Model: Why Prefill and Decode Differ

Most discussions of LLM inference treat it as a single workload with a single performance curve. This is the root cause of a class of sizing mistakes that no amount of GPU provisioning can fix - because they stem from a mismatch between what the hardware is good at and what the workload demands of it. A team that sizes for throughput on a decode-dominated workload and buys H100s for their TFLOPS will be disappointed - the bottleneck was never compute. More TFLOPS do not help when the chip is starving for memory bandwidth.

LLM inference is two distinct computational workloads running sequentially on the same hardware. They have different arithmetic profiles, hit different hardware limits, and respond to different optimizations. Understanding why requires one foundational concept: **arithmetic intensity**.

3.1 Arithmetic Intensity: The Number That Determines Your Bottleneck

Every GPU has two hard performance ceilings. The first is peak compute throughput, measured in TFLOPS - how many floating point operations the chip can execute per second. The second is peak memory bandwidth, measured in TB/s - how fast data can move between HBM (the high-bandwidth memory on the GPU) and the compute units that actually do the math.

Arithmetic intensity captures the ratio between these two demands for any given operation:

```
Arithmetic intensity = TFLOPs required / bytes of memory moved
```

When arithmetic intensity is high - many computations per byte fetched from memory - the operation is compute-bound: it saturates the TFLOPS ceiling before it saturates memory bandwidth. When arithmetic intensity is low - few computations per byte - the operation is memory-bandwidth-bound: the compute units sit idle waiting for data that memory cannot deliver fast enough.

The ridge point of the roofline is the arithmetic intensity at which both ceilings are hit simultaneously:

```
Ridge point = peak TFLOPS / peak HBM bandwidth (in FLOP/byte)
```

For an H100 SXM: approximately 989 TFLOPS (TF32) / 3.35 TB/s = roughly 295 FLOP/byte. Any operation above this threshold is compute-bound. Below it, memory bandwidth is the constraint. This single number is why prefill and decode require fundamentally different hardware thinking.

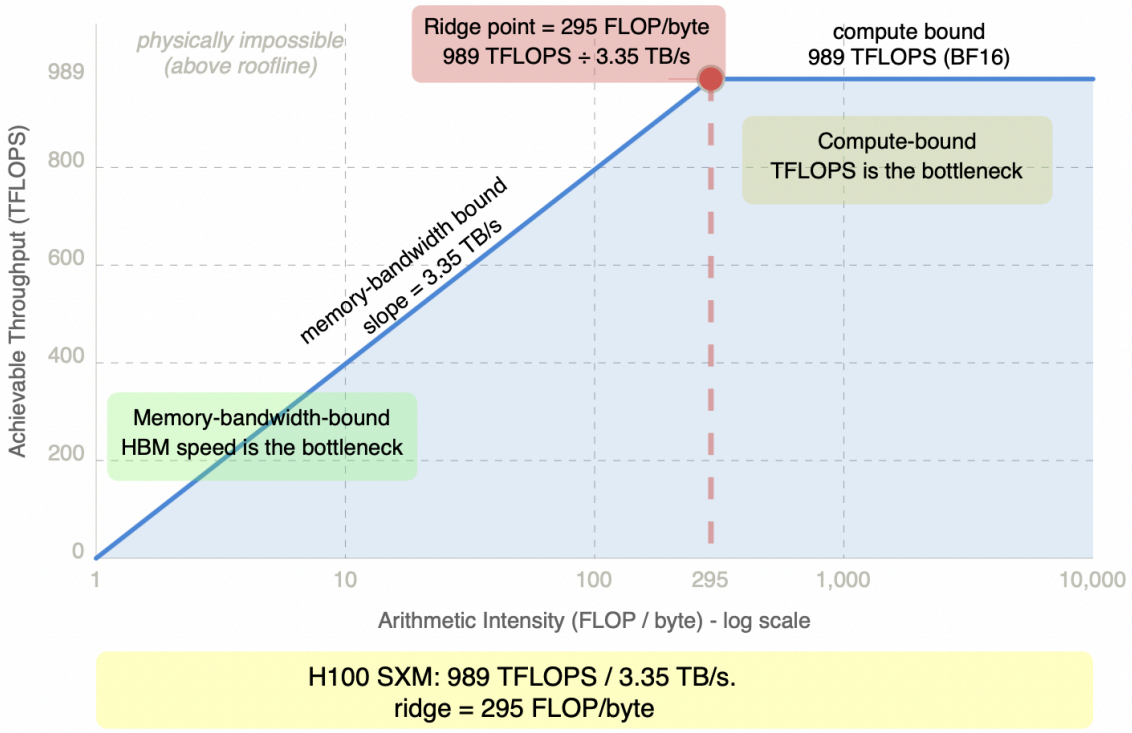

Figure 3.1 - The roofline model for H100 SXM
Every GPU operation is bounded by one of two hard ceilings. Below the ridge point (295 FLOP/byte on H100 SXM), memory bandwidth is exhausted before compute - adding more TFLOPS does nothing. Above it, compute is exhausted before memory bandwidth - faster HBM does nothing. The roofline is not a guideline: no software optimization can push an operation above it. Only restructuring the computation (e.g. larger batch sizes) or changing the hardware can shift where an operation sits on this chart. The next two sections show exactly where prefill and decode land.

The ridge point is the dividing line. The next two sections show exactly where prefill and decode land relative to it - and why they land on opposite sides.

3.2 Prefill Is Compute-Bound

Prefill lands well above the ridge point. Here is why. During prefill, the model processes all input tokens simultaneously in a single parallel forward pass. This is structurally a large matrix multiplication: the input token matrix (shape: sequence_length × hidden_dimension) is multiplied against the weight matrices in each transformer layer. Matrix multiplications have high arithmetic intensity because the same weight values are reused across many token positions in the same operation - the compute-to-memory-access ratio is high.

The practical consequence is significant: prefill saturates GPU compute even at batch size 1. A single request with a 4K-token prompt will fully occupy the CUDA cores of an H100. Adding more concurrent prefill requests to the same batch does not meaningfully increase per-request throughput - the compute ceiling is already being hit. What you gain from batching prefill is amortization of the fixed overhead per forward pass, not escape from the bottleneck. This means prefill throughput scales with TFLOPS, not with batching strategy.

The lever for higher prefill RPS is faster compute - either a higher-TFLOPS GPU or, for very large deployments, dedicated prefill instances separated from decode. This is the architectural motivation for prefill-decode disaggregation, covered in subsequent sections.

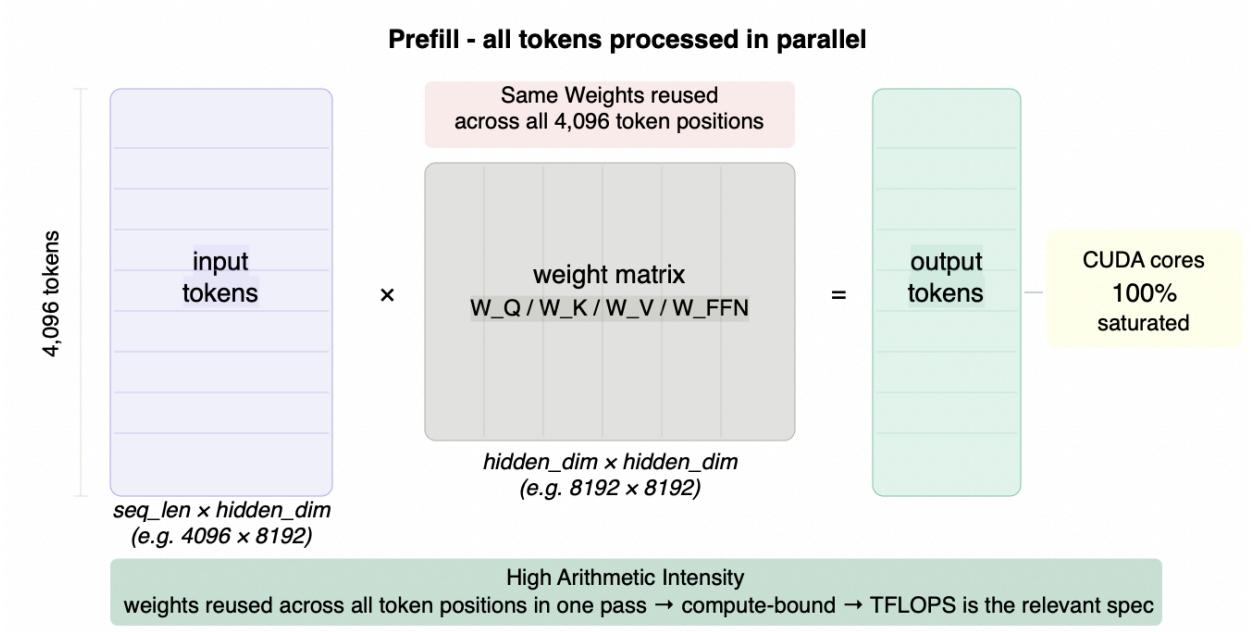

Figure 3.2 - Prefill: large matrix × large matrix
During prefill, all input tokens are processed simultaneously - a [seq_len × hidden_dim] matrix multiplied against each weight matrix in the layer stack. The same weights are reused across every token position in a single pass, producing high arithmetic intensity. CUDA cores stay fully occupied. The bottleneck is compute, not memory.

For prefill-dominated workloads - RAG pipelines processing long retrieved contexts, document summarization, legal or medical record analysis - the relevant GPU specification is TFLOPS. An H100 SXM at 989 TFLOPS BF16 dense completes the same prefill roughly 3× faster than an A100 at 312 TFLOPS BF16 - the memory bandwidth difference between them is proportionally much smaller and largely irrelevant for this workload type. More compute throughput means faster prefill, lower TTFT, higher prefill RPS. Memory bandwidth matters less because the arithmetic intensity of large matrix multiplications keeps the compute units occupied.

3.3 Decode Is Memory-Bandwidth-Bound

Decode is structurally the opposite problem.If prefill is the GPU doing too much work per byte, decode is the GPU doing too little - starved for data rather than overwhelmed by computation. Each decode step generates exactly one new token. To do so, the model must load the full weight matrix for every transformer layer from HBM into on-chip SRAM, perform a small matrix-vector multiplication (one token position against the full weight matrix), and discard the intermediate activations while updating the KV cache. Then repeat for the next layer.

The arithmetic intensity of this operation is extremely low. You are moving hundreds of gigabytes of weight data through the memory bus to perform a trivially small amount of

arithmetic on it - one token's worth of computation per multi-gigabyte weight matrix. The compute units finish their work almost instantly and then wait for the next chunk of weights to arrive from HBM. The GPU is not doing math most of the time. It is waiting for memory.

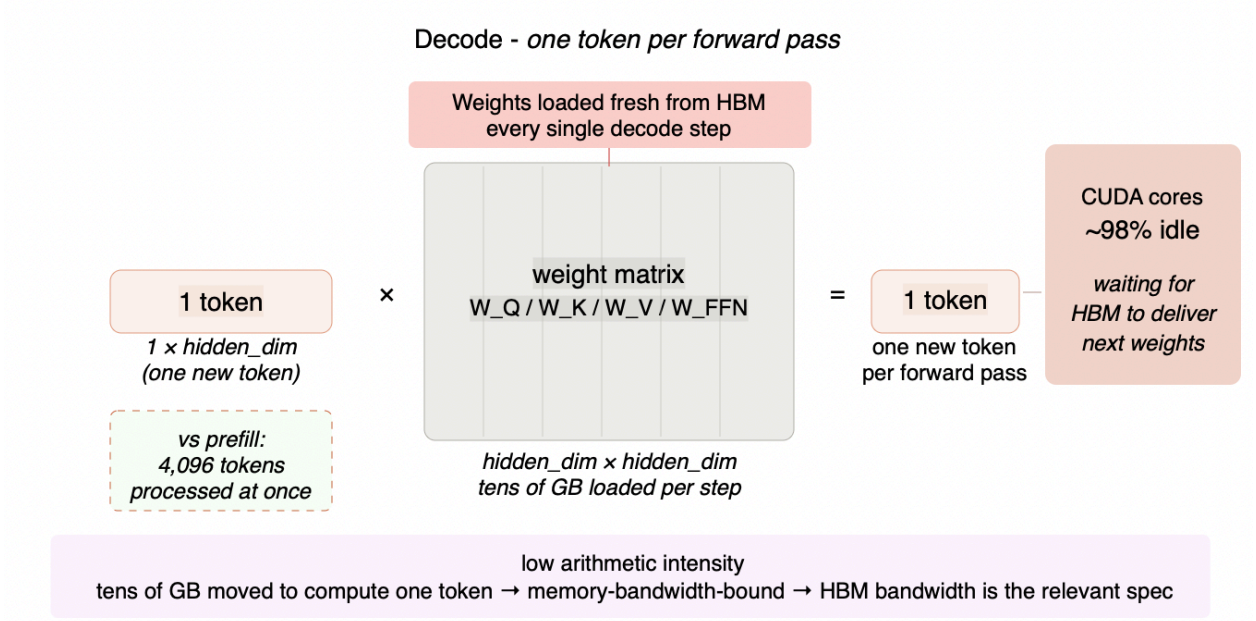

Figure 3.3 - Decode: large matrix × single vector
During each decode step, only one new token is generated. The full weight matrix - tens of gigabytes - is loaded fresh from HBM on every step, but multiplied against a single token vector. The arithmetic intensity collapses to near zero. CUDA cores finish their work almost instantly and sit idle while the memory bus delivers the next chunk of weights. The bottleneck is HBM bandwidth, not compute.

SARATHI's (chunks prefills and co-batches them with decodes so decodes piggyback on prefill compute) profiling data makes this concrete: at batch size 1, decode per-token cost can be up to 200× higher than prefill per-token cost on the same hardware. The weights do not change between a prefill step and a decode step - the computational structure does, and that structural change moves the operation from well above the roofline ridge point to well below it. This is not an inefficiency that can be tuned away. It is a direct consequence of arithmetic intensity collapsing from ~1,000 FLOP/byte during prefill to ~2 FLOP/byte during decode - a 500× difference in computational density on the same hardware running the same model.

For decode-dominated workloads - code generation, long-form content creation, extended reasoning chains - the relevant GPU specification is HBM bandwidth, not TFLOPS. The AMD MI300X at 5.3 TB/s and 192GB capacity delivers meaningfully better decode throughput than the H100 SXM at 3.35 TB/s and 80GB capacity for the same model, despite the H100 having superior raw compute. The comparison reverses for prefill-heavy workloads where TFLOPS dominates. This is not a marketing distinction - it is a direct consequence of where each workload sits relative to the roofline ridge point.

The roofline chart makes the contrast unavoidable. Placing both operating points on the same axes shows not just that prefill and decode are different - it shows *how*

different, and why the gap cannot be closed by any single hardware choice or software optimization.

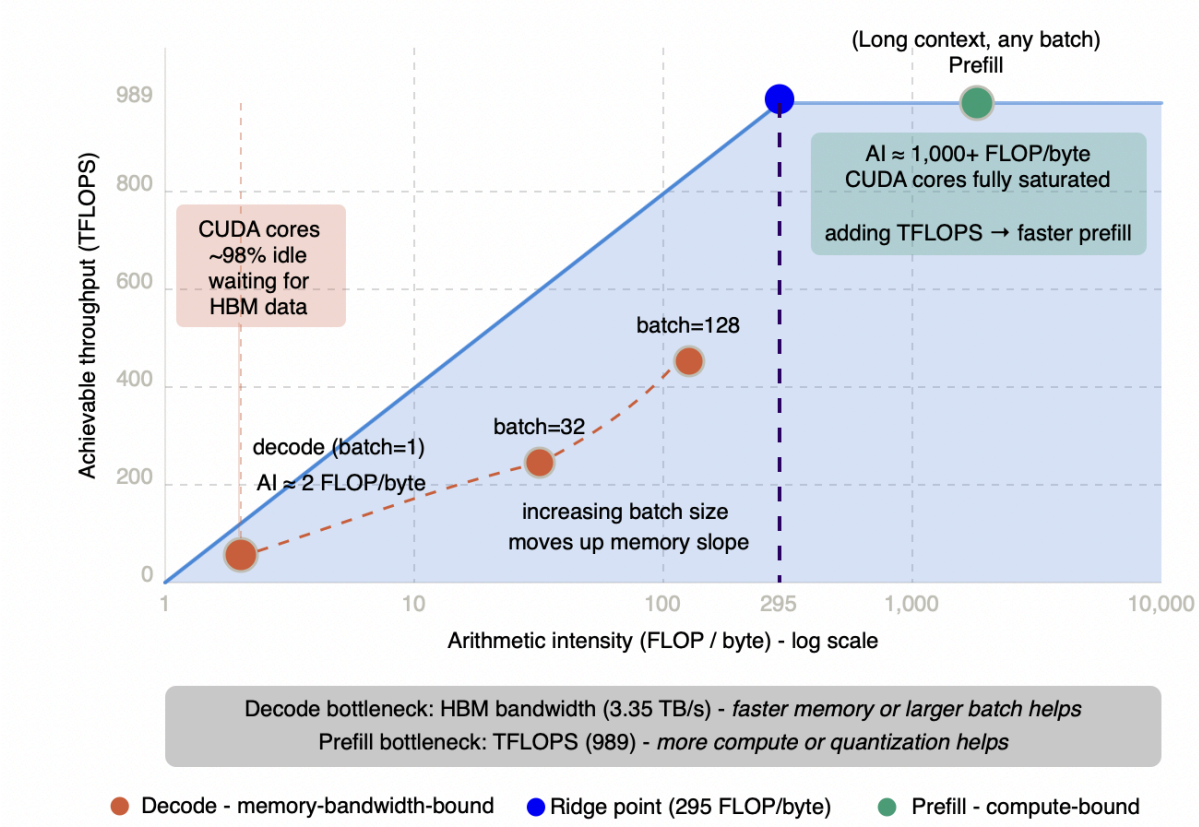

Figure 3.3.2 - Prefill and decode operating points on the H100 roofline
Prefill sits at the compute ceiling - processing all prompt tokens in parallel produces arithmetic intensity well above 295 FLOP/byte, fully saturating CUDA cores regardless of batch size. Decode at batch=1 sits deep in the memory-bandwidth-bound region at roughly 2 FLOP/byte, leaving CUDA cores ~98% idle while the memory bus works at capacity. Larger batch sizes move decode up the memory-bound slope - but never past the ridge point into compute-bound territory. The two phases are not variations of the same problem. They are different problems on the same hardware, and they respond to different solutions.

GPU selection for a mixed workload is therefore not a single decision. It is a weighted average of where your traffic sits on the prefill-decode spectrum. A RAG pipeline spending 80% of compute time in prefill should optimize for TFLOPS. A code generation service spending 80% of compute time in decode should optimize for HBM bandwidth. Buying the wrong GPU for your workload mix is a sizing mistake that no software configuration can compensate for. Batch size is the one lever that moves decode's operating point up the memory-bound slope - and it is significant enough to warrant its own section.

3.4 Batch Size Is the Primary Lever for Decode Efficiency

Batch size is the one variable that directly improves decode's arithmetic intensity - and therefore the one variable that moves its operating point up the memory-bound slope toward the roofline.

The mechanism is straightforward. Every decode step requires loading the full weight matrices for every transformer layer from HBM - a fixed bandwidth cost paid regardless of how many requests are being served simultaneously. At batch size 1, that cost buys one token of output. At batch size 64, the identical memory transfer produces 64 tokens of output. The arithmetic intensity scales linearly with batch size because the compute work multiplies while the memory access cost stays constant. This is the fundamental reason batching matters for decode in a way it does not for prefill: you are spreading a fixed bandwidth tax across more useful work.

The practical consequence is substantial. At batch=1 on a Llama-3 70B FP8 on H100, decode throughput is approximately 80 tokens per second - the memory bus is working at capacity but producing almost nothing because the compute units are nearly idle. At batch=128, throughput reaches approximately 4,700 tokens per second - roughly 60× higher - with the same weight-loading cost per step. The ceiling is a hardware limit, not a configuration limit. Once HBM bandwidth is fully saturated, no software optimization pushes the curve higher.

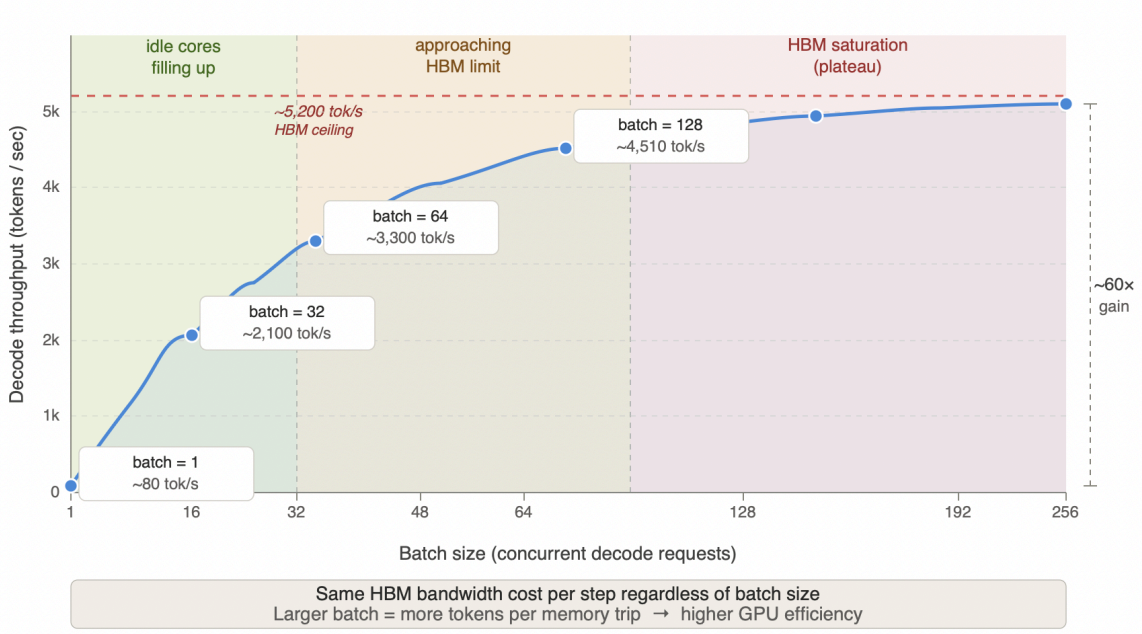

Figure 3.4 - Decode throughput vs batch size (Llama-3 70B FP8, H100 SXM)
At batch=1, the memory bus is already working at capacity - loading the full weight matrices every decode step - but producing only ~80 tokens per second because the compute units are nearly idle. Each additional request in the batch amortizes that same fixed memory cost across more simultaneous token generations, putting idle CUDA cores to work. Throughput scales near-linearly through the green zone as idle compute fills up, bends through the amber zone as the memory bus approaches saturation, and plateaus in the red zone when HBM bandwidth is fully exhausted - a

hardware ceiling no software optimization can push past. At batch=128 throughput reaches ~4,510 tok/s, roughly 60× higher than batch=1 for identical hardware cost per step. The plateau makes the constraint explicit: beyond ~192 concurrent requests, adding more users yields diminishing returns until more GPUs or faster HBM are introduced. This ~60× gap is why continuous batching is not optional for production deployments - a system serving requests one at a time is paying full HBM bandwidth cost for a fraction of the achievable output.

This asymmetry does not apply to prefill. Prefill already saturates compute at batch size 1 - processing all input tokens simultaneously naturally fills the CUDA cores. Adding concurrent prefill requests does not find idle compute capacity; it forces requests to share the same ceiling, increasing TTFT without a proportional throughput gain. The roofline positions of the two phases are fixed by their arithmetic intensity, and batching only helps the phase that has idle resources to fill. For decode, those resources are the compute units. For prefill, there are none to recover. This structural difference is the architectural motivation for separating them entirely - covered in subsequent sections.

3.5 Hardware Strategy: The Roofline Trade-off

Batch size moves the operating point up the memory-bound slope - but it cannot change which side of the ridge point a workload sits on. That is a hardware decision, and the roofline model makes it precise.

A fleet optimized for prefill needs TFLOPS. The H100 SXM at 989 TFLOPS TF32 (1,979 TFLOPS BF16 with sparsity) and the B200 at 2,250 TFLOPS BF16 dense (4,500 TFLOPS with sparsity) are the right choices for RAG pipelines, document summarization, and any workload where input context is long and output is short. Memory bandwidth is secondary - the arithmetic intensity of large matrix multiplications keeps the compute units saturated regardless.

A fleet optimized for decode needs HBM bandwidth. The AMD MI300X at 5.3 TB/s versus the H100 SXM at 3.35 TB/s delivers approximately 58% higher peak memory bandwidth - a direct consequence of the memory bus being the binding constraint for decode. The MI300X also carries 192 GB of HBM versus 80 GB on the H100, meaning larger models or higher KV cache budgets without tensor parallelism. NVIDIA's own H200 at 4.8 TB/s and 141 GB HBM3e partially closes this gap and is the current recommended Hopper-generation upgrade for decode-heavy workloads. For code generation, extended reasoning, and chat workloads where output length dominates, optimizing for bandwidth per dollar rather than TFLOPS per dollar is the correct framing.

For mixed workloads, neither GPU is unambiguously correct. The right answer is to measure your actual input-to-output length ratio under production traffic, compute the fraction of wall-clock time spent in prefill versus decode, and weight the hardware comparison accordingly. Headline throughput benchmarks are almost always measured under prefill-favorable conditions and will overstate performance on decode-heavy workloads.

The architectural implication. The roofline analysis exposes a fundamental tension: any single GPU fleet is a compromise between two workloads with opposite hardware requirements. The resolution is disaggregated prefill-decode architecture - separate GPU

pools for each phase, each sized and configured for its specific bottleneck. This is covered in the disaggregation section later in this guide.

Quantization affects the two phases differently. Reducing weight precision from FP16 to FP8 or INT8 halves the bytes transferred per weight matrix load. For decode - where the bottleneck is memory bandwidth - this directly raises the throughput ceiling: the same HBM bandwidth now moves twice as many weight values per second, and FP8 decode throughput approaches 2× FP16 at bandwidth saturation. For prefill - where the bottleneck is compute - the same quantization saves VRAM and reduces weight-loading time but does not increase TFLOPS. The speedup for prefill comes primarily from fitting larger batches or longer contexts into the same memory budget, not from hitting a higher compute ceiling. This asymmetry means quantization decisions cannot be evaluated on a single throughput number - they must be evaluated separately for each phase. Section 4 covers quantization strategy in full.

4 - Quantization: Trading Precision for Scale

Section 3 established that quantization affects prefill and decode differently - halving memory bandwidth pressure for decode while primarily freeing VRAM capacity for prefill. Before examining those phase-specific effects, the foundation is the memory arithmetic, because that is where the leverage is most visible.

Neural network weights follow smooth, well-behaved distributions clustered around zero. Representing each weight as a 16-bit float when an 8-bit or 4-bit integer captures nearly the same information is like printing a temperature forecast to six decimal places - the extra precision costs memory and bandwidth without meaningfully changing the answer. The practical stakes are significant: Llama-3 70B at FP16 requires 140 GB for weights alone. The same model at INT4 requires 35 GB - the difference between a multi-node setup and a single 8-GPU node with room left for KV cache. At production scale, quantization is not a compromise for resource-constrained environments. It is the standard operating mode.

No other single optimization lever simultaneously reduces model weight memory, shrinks KV cache footprint, increases HBM bandwidth efficiency, and improves decode throughput - without changing model architecture or retraining from scratch. The sections that follow examine each of these effects in turn.

4.1 Weight-Only Quantization: Compress the Model, Keep Everything Else

Weight-only quantization compresses the static model weight matrices while leaving activations and KV cache at higher precision. Think of it like compressing a large file before transmission - the weights are stored and loaded in compressed form, then decompressed for computation. The KV cache and runtime activations are unaffected.

Three formats dominate production deployments:

GPTQ (Generalized Post-Training Quantization) compresses model weights to lower precision by solving for the quantized values that best preserve what each layer actually *produces* - not just what individual weights look like in isolation. This distinction matters. Naive quantization rounds each weight independently to the nearest representable lower-precision value. GPTQ does something smarter: it runs a small representative sample of real inputs - called a *calibration dataset* - through the model, observes how each layer transforms those inputs, and then chooses quantized weights that minimize the error in the layer's *output* rather than the error in the weights themselves.

The optimization runs block by block within each layer. As weights are quantized in each block, GPTQ measures how much the layer's output has drifted from the original and uses that information - specifically a mathematical structure called the *Hessian*, which captures

how sensitive the output is to changes in each weight - to distribute the quantization error as evenly as possible across the remaining weights in the block. Weights that the model is highly sensitive to get quantized more carefully; weights that have little influence on the output absorb more of the rounding error. The result is a quantized model where accumulated output error is minimized globally across each layer, not just locally weight by weight.

No retraining is required - GPTQ runs once after training is complete, typically in a few GPU-hours for a 70B model. It delivers strong output quality even at INT4 and is widely supported across vLLM, TensorRT-LLM, and HuggingFace.

AWQ (Activation-aware Weight Quantization) takes a sharper approach than GPTQ. It starts from an observation: not all weights contribute equally to model output. When a weight is paired with a high-magnitude activation - a large intermediate value flowing through the network - even a small quantization error in that weight gets amplified significantly in the layer's output. AWQ identifies the roughly 1% of weights in this high-sensitivity category and protects them from aggressive quantization, while compressing the remaining 99% more aggressively than it otherwise would. The result is better accuracy than GPTQ at the same bit width, particularly on reasoning and instruction-following tasks, because the weights that actually drive output quality are preserved at higher precision. For production serving at INT4, AWQ is generally the preferred choice.

GGUF, used by llama.cpp, applies mixed-precision quantization per layer and is the standard format for on-device and CPU inference. It is outside the scope of cloud-scale serving covered in this guide but worth knowing for edge deployment scenarios.

The memory impact is straightforward: FP16 to INT8 reduces weight memory by 50% with negligible quality loss on most benchmarks. FP16 to INT4 via AWQ or GPTQ reduces it by 75% - a 1-3% perplexity increase that is generally imperceptible in production chat or RAG workloads but may surface in complex multi-step reasoning chains.

One important caveat: weight-only quantization captures the weight saving entirely but leaves KV cache and activation memory unchanged. At low concurrency this is sufficient - weights dominate the budget. At high concurrency with long context, KV cache grows to match or exceed weight memory, and weight-only quantization captures a shrinking fraction of the total saving. This is why KV cache quantization is a separate lever, covered in the next subsection.

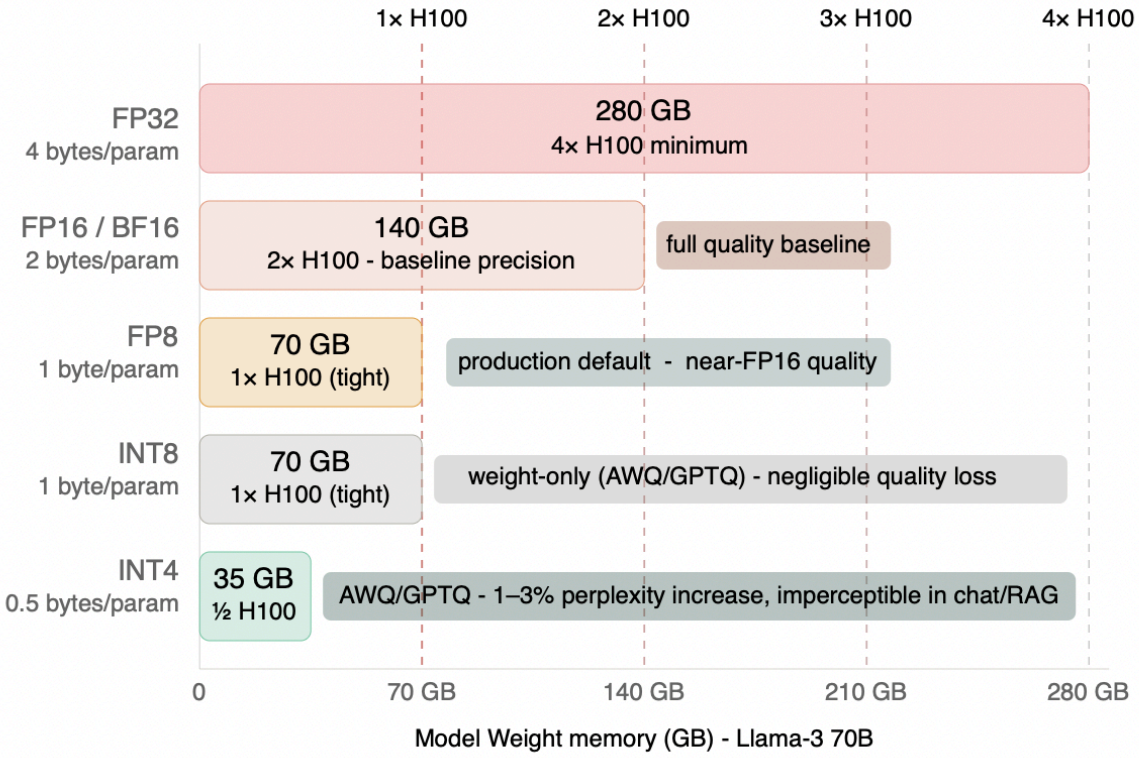

Figure 4.1 - Model weight memory by precision format: Llama-3 70B
Every halving of bit-width halves the weight memory footprint. FP16 to FP8 cuts 140 GB to 70 GB - the difference between needing 2 H100s and fitting on 1. FP16 to INT4 (AWQ/GPTQ) cuts to 35 GB - a 75% reduction with typically imperceptible quality loss for chat and RAG workloads. FP8 is the recommended production default: it unlocks native Tensor Core acceleration on H100 and Blackwell while preserving near-FP16 output quality. Note that these figures cover model weights only - KV cache and activation memory are calculated separately and follow their own precision rules.

4.2 Activation Quantization: Unlock Hardware Acceleration

Weight-only quantization, as covered in the previous section, compresses the weights but leaves activations untouched - and critically, it decompresses weights back to FP16 before the actual matrix multiplication runs. This means the compute path stays in FP16 regardless of how aggressively the weights were compressed. Activation quantization changes this. By keeping both weights and activations in low precision all the way through the computation, it unlocks a different category of hardware capability entirely.

Modern GPUs like the H100 and Blackwell have dedicated hardware units - Tensor Cores - that run INT8 and FP8 matrix multiplications significantly faster than FP16. Weight-only quantization cannot use these faster units because the activations are still in FP16. Activation quantization can. This makes activation quantization a compute throughput lever, not just a memory lever - and that distinction matters for decode-heavy workloads where throughput is the binding constraint.

The catch is that activations are harder to compress than weights. Weights are fixed after training and follow smooth, predictable distributions - they compress cleanly. Activations

change with every input, and they regularly produce large outlier values far outside the normal range. Low-bit formats like INT8 struggle here. You can either clip the outliers and lose information, or stretch the representable range to cover them and waste most of your precision on values that rarely occur. Either way, naive INT8 activation quantization hurts model quality noticeably on transformer models.

SmoothQuant gets around this with a neat mathematical trick. If you multiply a weight matrix by some scaling factor S and divide the corresponding activation by the same S, the final output of that layer does not change - the two operations cancel out exactly. SmoothQuant uses this to move the difficulty from activations to weights. It picks scaling factors that shrink activation outliers down to a range INT8 can handle cleanly, and bakes the inverse of those scaling factors into the weights to keep the math identical. This is done once, offline, after training. At inference time, activations are well-behaved and easy to quantize, the weights already carry the compensation, and the output is mathematically equivalent to the original. No retraining needed.

Think of it like balancing a seesaw. If one side is too heavy to fit within a weight limit, you do not remove weight - you slide the fulcrum. The total weight is unchanged, the balance is unchanged, but now both sides sit within the allowed range. SmoothQuant slides the fulcrum between activations and weights so that the activation side - which was too heavy for INT8 - becomes manageable, while the weight side absorbs the shift. The total computation is identical.

The result is a modified set of weights and scaling factors that make activations much smoother and easier to quantize at runtime.

FP8 (W8A8) is where most production systems land today on H100 and Blackwell. The notation is simple: W means weights, A means activations, and the number is the bit width. W8A8 means both weights and activations run in 8-bit precision. You will also see W4A16 - 4-bit weights, 16-bit activations - which is weight-only quantization. This naming convention appears throughout vLLM, TensorRT-LLM, and most inference benchmarks and is worth internalizing.

To understand why FP8 is preferred over INT8 at the same bit width, it helps to think of the three formats as a spectrum. FP16 is accurate but heavy - 2 bytes per value, high memory and bandwidth cost. INT8 is fast and light - 1 byte per value - but uses a fixed-point representation that spreads 256 evenly-spaced values across a fixed range. That works well for weights but is rigid when activations have occasional large spikes, which is the normal condition in transformer inference. FP8 sits deliberately in the middle: it uses floating-point representation like FP16, allocating bits between a range exponent and a precision mantissa, which gives it a much wider dynamic range than INT8 at the same 1-byte storage cost. In practice FP8 handles activation outliers naturally, typically needs no SmoothQuant preprocessing, and preserves near-FP16 output quality. On H100 SXM, FP8 Tensor Cores deliver roughly 2× the throughput of FP16 at equivalent batch sizes - making FP8 W8A8 the default for high-throughput production inference on current hardware.

FP8 W8A8 solves the compute path. The remaining memory problem - the KV cache, which at high concurrency can grow larger than the model weights themselves - has its own quantization lever, and it is the simplest one to apply in practice.

4.3 KV Cache Quantization: The Easiest Win in Production

This is the simplest quantization lever in the stack. Weight quantization requires careful method selection, calibration datasets, and accuracy validation that can take days. KV cache quantization requires almost none of that - in vLLM it is a single startup flag: `kv_cache_dtype=fp8`. TensorRT-LLM has an equivalent parameter at engine build time. The reason it is so forgiving is that small errors in cached key-value vectors rarely propagate strongly enough to affect final token probabilities in a meaningful way - the attention mechanism is naturally tolerant of small perturbations in the KV vectors.

The payoff is immediate and concrete. Reducing KV cache precision from FP16 to FP8 cuts KV cache memory in half. Half the KV cache memory means you can fit twice as many concurrent requests on the same hardware at the same context length. This is not a traditional throughput optimization - it is a concurrency gain, which translates directly into higher requests per second per GPU without touching the model or changing the system architecture.

Think of it like a warehouse that stores parcels. If you switch from large boxes to smaller ones that hold the same contents, you can fit twice as many parcels in the same space. The parcels - your KV vectors - are slightly less precisely packed, but they still contain everything needed to do the job.

On newer hardware like NVIDIA Blackwell, even more aggressive formats are available. NVFP4 for KV cache reduces memory by another ~50% compared to FP8, with negligible accuracy loss - typically under 1% degradation across long-context and code generation benchmarks. If FP8 doubles your concurrent request capacity over FP16, NVFP4 doubles it again - a 4× concurrency multiplier on Blackwell hardware, from a configuration change alone.

For long-context workloads or high-concurrency serving targets, KV cache quantization is almost always the first optimization to reach for. No retraining, no architecture changes, no calibration pipeline - just a configuration update, a validation run on your target workload, and meaningfully more capacity on the same hardware.

4.4 Sparsity: A Different Dimension of Compression

Quantization reduces how precisely each weight is stored. Sparsity takes a different approach - it removes weights entirely by setting many of them to zero. Think of it like cleaning up a workspace: quantization repacks everything into smaller containers, while sparsity throws away items you do not actually need. Once a weight is zeroed out, the hardware does not have to load or compute it at all.

Modern GPUs like the H100 support a specific pattern called 2:4 structured sparsity - out of every 4 consecutive weights, exactly 2 must be zero. Because this pattern is fixed and

predictable, the hardware can skip the zero values during matrix multiplication and deliver up to 2× the compute throughput compared to a dense model of the same size. Tools like SparseGPT can convert a trained dense model into this format without retraining, by identifying which weights can be safely zeroed with minimal impact on output quality.

The practical caveat matters for sizing decisions. The 2× compute speedup from sparsity is most valuable when your workload is compute-bound - large batch prefill, for example. For decode at small batch sizes, the GPU is memory-bandwidth-bound, and removing weights does not help because the bottleneck is data movement, not arithmetic. Sparsity does reduce the number of weights loaded from HBM per step, which provides some memory bandwidth benefit, but the gains are less dramatic than in the compute-bound regime. If your primary bottleneck is decode throughput at low concurrency, sparsity is less impactful than KV cache quantization or batching strategy.

Sparsity and quantization can be combined - a sparse FP8 model benefits from both fewer weights and lower precision per weight. This is increasingly common on H100 and Blackwell for prefill-heavy workloads where maximizing compute throughput is the goal. For most production deployments, the recommended order of operations is: quantize first to FP8, validate quality, then evaluate sparsity if compute throughput remains the binding constraint after quantization.

4.5 Choosing the Right Approach

The four techniques covered in this section are not mutually exclusive - they stack. The question is not which one to use, but in what order and combination for your specific workload. A practical starting rule: begin with KV cache quantization because it requires no accuracy validation and delivers immediate concurrency gains, then layer weight quantization on top if memory or throughput constraints remain. The decision table below maps the most common production scenarios to the right starting point.

Scenario	Recommendation	Why
Maximum output quality, VRAM not constrained	FP16 or BF16 (no quantization)	Highest fidelity, no approximation
Standard production serving on H100 / Blackwell	FP8 (W8A8) + FP8 KV cache	Best balance of accuracy, throughput, and memory - native Tensor Core support on current hardware
Decode-dominated workload, throughput is the bottleneck	FP8 W8A8 + FP8 KV cache + maximize batch size	Decode is memory-bandwidth-bound - FP8 halves bytes moved per step, directly raising the throughput ceiling

Memory constrained, quality still important	INT8 weights (AWQ / GPTQ) + FP16 KV cache	Reduces model weight memory while preserving activation and KV cache precision
Maximum compression, quality-tolerant workloads (chat, RAG)	INT4 via AWQ + FP8 KV cache	75% weight memory reduction with typically imperceptible quality loss for chat and RAG workloads
Long context or high concurrency is the primary bottleneck	Prioritize KV cache quantization first - FP8, or NVFP4 on Blackwell	Directly halves or quarters KV cache memory, doubling or quadrupling concurrent request capacity
Unsure where to start	KV cache quantization before weight quantization	Fastest and safest first step - no calibration, no accuracy validation pipeline, immediate capacity gain

One column worth internalizing across all rows: the "why" is always rooted in which hardware resource is the binding constraint for your workload. Memory-bound scenarios benefit most from reducing bytes. Compute-bound scenarios benefit most from unlocking faster Tensor Core paths. Matching the technique to the actual bottleneck - rather than applying quantization generically - is what separates a principled sizing decision from a guess.

Quantization addresses the per-GPU memory and compute efficiency problem. The next lever - parallelism strategy - addresses how to scale across multiple GPUs once the single-GPU operating point is optimized.

5 - Parallelism Strategy: Splitting a Model Across GPUs

The single-GPU operating point is now optimized. The next question is what happens when one GPU is not enough - either because the model does not fit in memory, or because a single device cannot meet your throughput targets even when it fits. A 405B model at FP8 requires well over 200 GB - multi-GPU execution is not optional. Even models that fit on one GPU may fall short of RPS targets under production concurrency. Parallelism is how you scale beyond both limits.

There are four primary parallelism strategies for LLM inference. They differ in what they partition, what they optimize for, and the communication overhead they introduce. Choosing the wrong strategy - or the right strategy on the wrong interconnect - can negate the theoretical gains entirely. Tensor Parallelism=4 on PCIe, for example, can deliver worse latency than Tensor Parallelism=2 on NVLink despite using twice the hardware, because the all-reduce overhead consumes more time than the compute saving buys back. The interconnect is not a footnote - it is a first-class input to the parallelism decision.

5.1 Tensor Parallelism: Split the Math, Reduce the Latency

Tensor parallelism splits weight matrices within a layer across GPUs. Each GPU holds a fraction of each weight matrix, computes a partial result on every forward pass, and an all-reduce operation combines those partial results before the computation moves to the next layer. Every GPU participates in every layer of every request.

The benefit is lower latency: each layer's compute is distributed, reducing per-step wall-clock time. TP=4 on a 70B model reduces per-GPU memory to approximately 35 GB and shortens each layer's matrix multiplication proportionally - four GPUs doing a quarter of the work each, in parallel.

The cost is communication. Each layer requires an all-reduce across GPUs, making interconnect bandwidth critical. On NVLink (~900 GB/s), this overhead is modest; on PCIe (~128 GB/s), it can consume 20-30% of step time, significantly reducing the gains. In practice, TP above 2 without NVLink rarely delivers the expected improvement and often makes things worse.

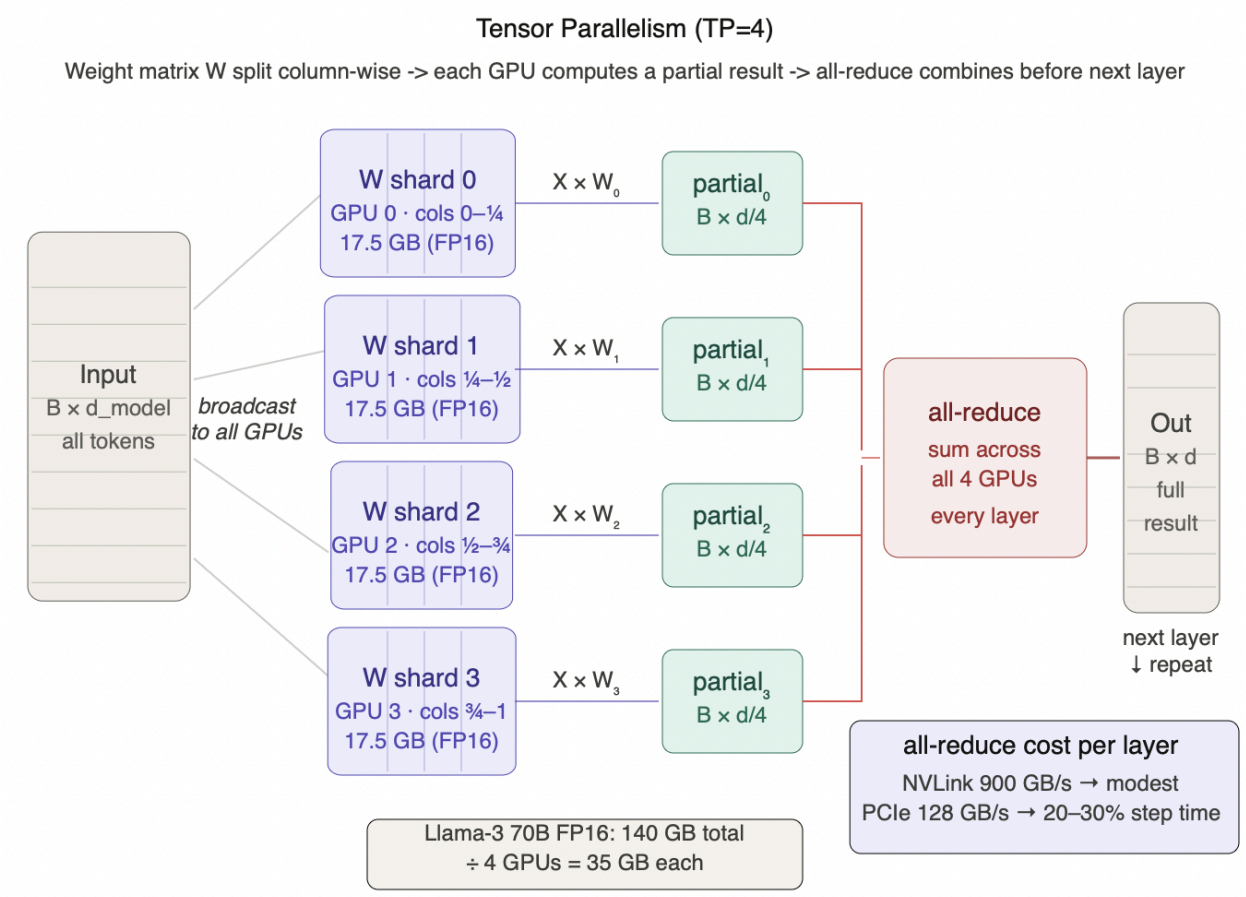

Figure 5.1 - Tensor parallelism (TP=4): one transformer layer
The weight matrix W is split column-wise across 4 GPUs - each GPU holds one quarter of the columns (17.5 GB for Llama-3 70B FP16). All 4 GPUs receive the full input batch simultaneously (broadcast), each computes its partial result independently in parallel, then a single all-reduce sums the four partial outputs before the result passes to the next layer. This repeats at every layer boundary. The memory benefit is immediate - 140 GB drops to 35 GB per GPU. The cost is the all-reduce: trivial on NVLink (900 GB/s) but 20-30% of step time on PCIe (128 GB/s), which is why TP > 2 without NVLink rarely delivers expected latency improvements.

Tensor parallelism also directly reduces the number of independent model replicas per node. At TP=4 on an 8-GPU node you have 2 replicas instead of 8. At TP=8 you have one replica per node - minimum latency, maximum per-request speed, but a single node can handle far fewer concurrent requests than a lower-TP configuration. This is the fundamental tradeoff: TP trades aggregate throughput for per-request latency.

For online serving, use the lowest TP degree that fits the model in memory and meets your per-GPU memory budget. More replicas at lower TP typically deliver higher aggregate throughput. The exception is strict TTFT SLOs - if latency requirements cannot be met by a single GPU at the required concurrency, higher TP is justified despite the throughput cost.

5.2 Pipeline Parallelism: Split the Layers, Watch for Bubbles

Tensor parallelism distributes the compute within each layer. Pipeline parallelism takes a different approach entirely - it distributes the layers themselves.

Pipeline parallelism partitions the model's layers across GPUs sequentially rather than splitting individual layers across GPUs. GPU 0 processes layers 1-10, passes the output activations to GPU 1 which processes layers 11-20, and so on. Communication is simple point-to-point transfer of activations between adjacent GPUs - much smaller than the all-reduce operations in tensor parallelism. This makes PP more communication-efficient than TP at scale, particularly for very large models distributed across multiple nodes where inter-node bandwidth is limited. It also scales memory cleanly: with PP=4, each GPU holds one quarter of the model's layers and one quarter of the weight memory.

The fundamental problem with PP for online serving is pipeline bubbles. When a request enters the pipeline, GPU 0 processes its layers and passes activations to GPU 1 - but GPU 0 is now idle until the next micro-batch arrives. GPUs at the front and back of the pipeline are idle whenever they are waiting for upstream or downstream stages. At small batch sizes these bubbles dominate. At large batch sizes with micro-batching, bubbles shrink as a fraction of total compute time but never disappear entirely.

```
Bubble fraction = (PP − 1) / (m + PP − 1)
```

where m is the number of micro-batches. To halve the bubble fraction at PP=4, you need to roughly double the micro-batch count - which means holding twice as many micro-batches worth of activations in memory simultaneously. The memory cost of bubble reduction is real and must be factored into your sizing.

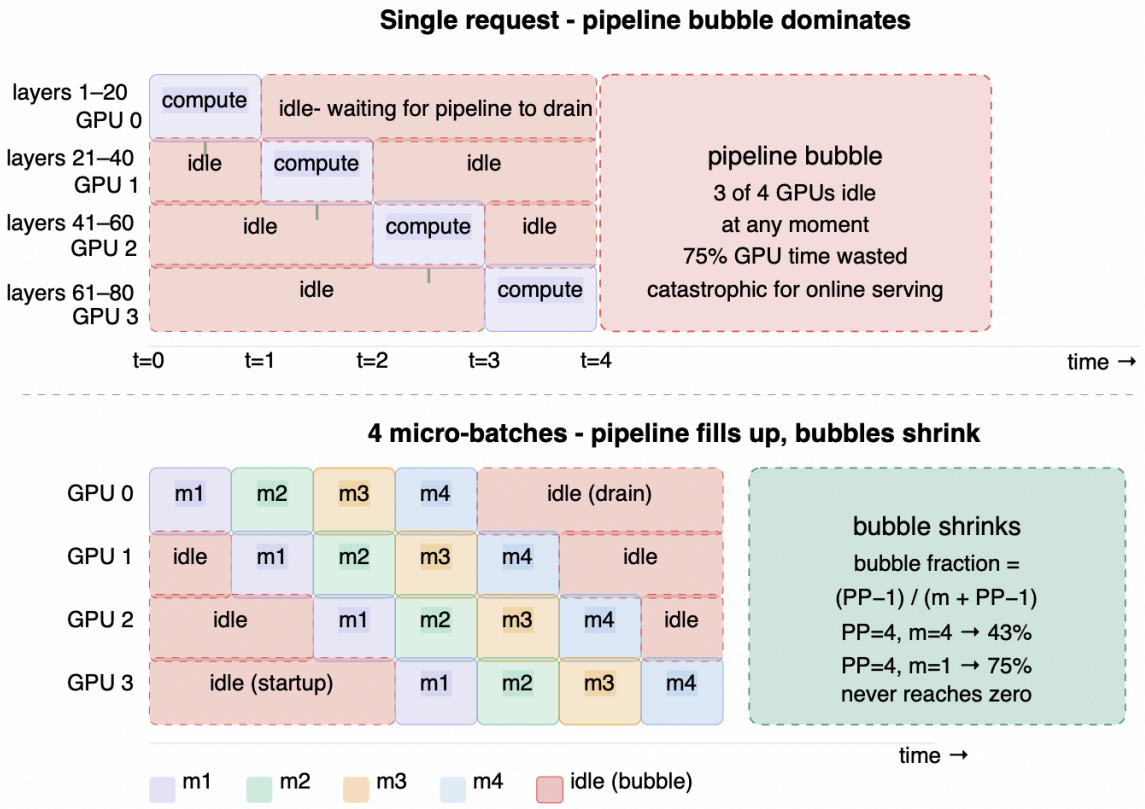

Figure 5.2 - Pipeline parallelism (PP=4): the bubble problem
Top: a single request enters the pipeline - GPU 0 computes its layers and passes activations down, then sits idle for 3 time steps while GPUs 1, 2, and 3 process sequentially. At any given moment 3 of 4 GPUs are idle - 75% of GPU time wasted. Bottom: 4 micro-batches fill the pipeline - while GPU 3 processes m1, GPU 2 processes m2, GPU 1 processes m3, and GPU 0 processes m4 simultaneously. Bubbles shrink to the startup and drain edges. But the bubble fraction formula (PP−1)/(m + PP−1) shows it never reaches zero: PP=4 with 4 micro-batches still wastes 43% of GPU time. This is why PP is reserved for models too large to fit within a single NVLink domain - for online serving with variable input lengths, TP within a node is always preferred.

Pipeline parallelism is best suited for offline batch inference where you can keep the pipeline full with large batches, and where per-request latency matters less than aggregate throughput. For latency-sensitive online serving with variable input lengths - which LLM workloads almost always have - TP is generally preferred within a node, with PP reserved for cases where the model simply cannot fit within a single node's NVLink domain.

When a model exceeds a single node's NVLink domain entirely, neither TP nor PP alone is sufficient - which is where the two strategies are combined, covered next.

5.3 Data Parallelism: The Cleanest Scaling Strategy

Tensor and pipeline parallelism both solve the same class of problem - fitting a model that is too large or too slow for one GPU. Data parallelism solves a completely different problem: what do you do when your single instance is already well-configured but you simply need more of them?

Data parallelism runs N completely independent model replicas across N GPU groups. Each replica processes its own request stream with no inter-replica communication during inference - no all-reduce, no activation transfer, no coordination overhead of any kind between replicas at the compute level. A load balancer distributes incoming requests across replicas. Doubling the replica count doubles RPS capacity with linear cost scaling and zero latency overhead. The only hard constraint is that each GPU group must hold a complete model copy - DP does not reduce per-instance memory requirements.

This simplicity is real but it comes with a few operational nuances that production deployments must handle explicitly.

Load balancing is not round-robin. LLM requests have wildly different token lengths and therefore wildly different latency profiles. Naive round-robin routing ignores replica load state - a long-context request routed to an already-loaded replica causes head-of-line blocking that degrades the entire replica's queue. Production deployments use latency-aware or queue-depth-aware routing. The load balancer is a first-class component of the DP architecture, not an afterthought.

Cold start latency scales with model size. Each replica is a full model copy. At 70B FP8 that is 70 GB per replica - loading from NVMe to GPU before the replica can serve its first request. Autoscaling response time for DP is directly proportional to model size, which means large models have slow autoscaling. Size down aggressively with quantization before

scaling out with DP, and pre-warm replicas where possible rather than spinning them up on-demand.

Multi-turn sessions require sticky routing. Each replica maintains its own independent KV cache. A multi-turn conversation where subsequent requests land on different replicas will find no cached KV state from prior turns - the model effectively loses context. Production deployments either implement sticky routing to pin sessions to replicas, or accept the KV cache miss cost and recompute context on each turn. The choice has both latency and cost implications.

Rolling deployments require replica-level coordination. Replicas share the same weights at any given time, but updating model versions across a fleet requires careful replica management - canary deployments, A/B version splits, and graceful drains all need orchestration at the replica level. DP replicas are independent at the compute level but not at the operations level.

The production scaling pattern that emerges from all of this: use TP within a node to fit the model and meet latency SLOs, then scale horizontally with DP across nodes to meet throughput requirements. TP handles the memory and latency problem. DP handles the capacity problem. These two levers compose cleanly at the architecture level - the operational complexity lives in the DP layer, not the TP layer.

The three strategies above apply to dense transformer models - every parameter participates in every forward pass. Mixture-of-Experts architectures break that assumption entirely, and when they do, a fourth parallelism strategy becomes not just useful but necessary.

5.4 Expert Parallelism: For MoE Architectures

In every parallelism strategy covered so far, every parameter participates in every forward pass. MoE architectures break this assumption - and when they do, the economics of inference change fundamentally.

Mixture-of-Experts models - DeepSeek-V3, Mixtral 8×7B, Qwen-MoE - have changed what "model size" means for inference. Each token is routed to a small subset of specialist feed-forward layers called experts, with the rest unused for that token. Total parameter count is large; active parameter count per forward pass is much smaller. DeepSeek-V3 has 671B total parameters but activates only ~37B per token - you store a 671B model but pay compute cost closer to a 37B model on every forward pass. The economic leverage is real, but it comes with a set of engineering problems that dense parallelism strategies do not have.

How EP works. Expert Parallelism distributes different experts across different GPUs. When a token is routed to Expert 7, it is dispatched via an all-to-all communication to the GPU hosting Expert 7's weights, computed there, and the result returned via a second all-to-all. This allows serving models with hundreds of billions of parameters at a fraction of the compute cost implied by their total size - because most of those parameters are inactive for any given token.

The engineering challenge is routing overhead: tokens must be dispatched to the correct expert GPU, which requires fast all-to-all communication across the expert GPU pool. Wide Expert Parallelism (Wide-EP), implemented in TensorRT-LLM, scales this to very large expert pools and has demonstrated 1.8x higher per-GPU throughput than smaller EP setups on an NVL72 system. Wide-EP works because NVL72's 72-GPU NVLink domain provides the all-to-all bandwidth density needed to keep routing latency from dominating - on systems without this interconnect density, Wide-EP's gains shrink significantly. For MoE models at scale, EP is not optional - it is the mechanism that makes the economics viable.

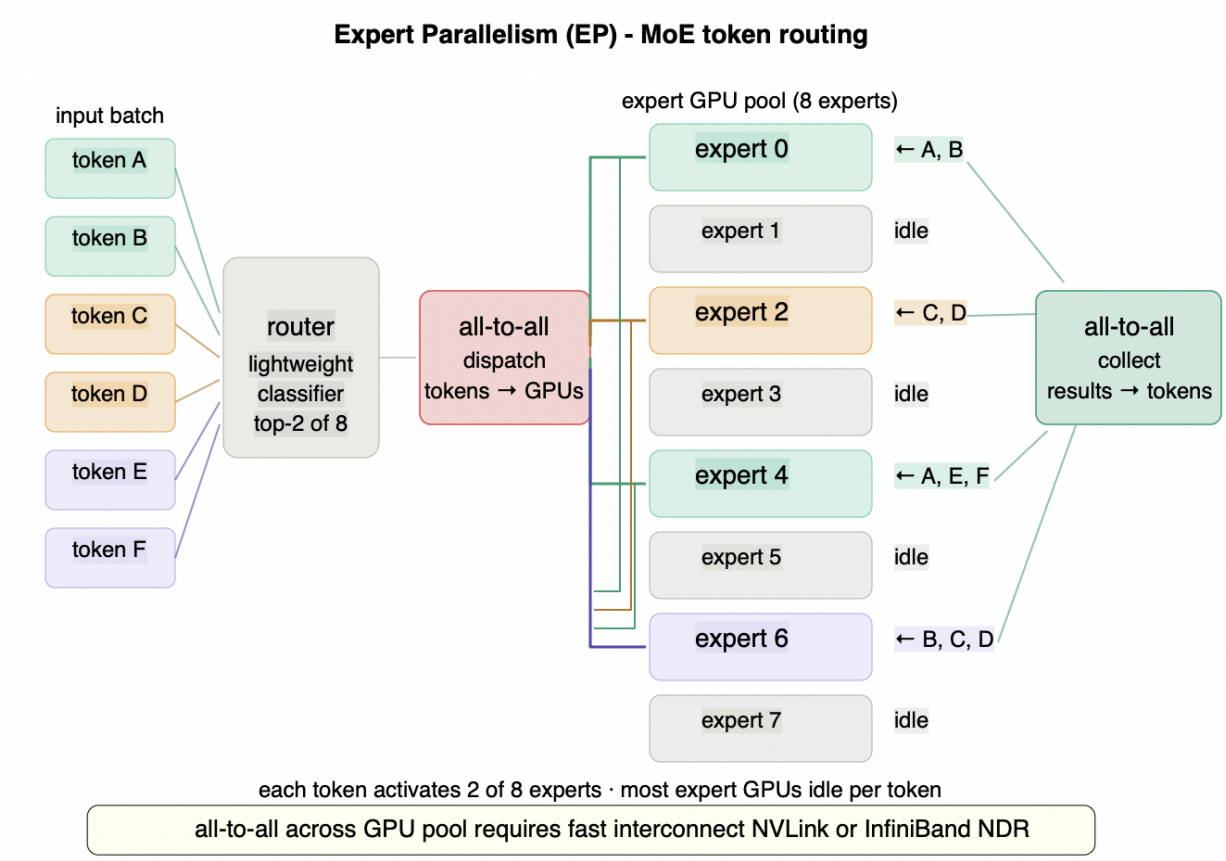

Figure 5.4 - Expert parallelism: MoE token routing across GPU pool
Each token is scored by a lightweight router and dispatched to its top-2 experts out of the full expert pool. An all-to-all communication sends each token to its assigned expert GPUs, each expert computes independently, and a second all-to-all collects results back to the originating tokens. At any given moment most expert GPUs are idle - for DeepSeek-V3 with 256 experts, each token activates only 2, leaving 254 inactive. The economic insight: you store 671B parameters but pay compute cost for only ~37B active parameters per token. The contrast with TP is fundamental - TP broadcasts every token to every GPU; EP routes each token to a small subset. The engineering challenge is the all-to-all communication pattern, which requires fast interconnect (NVLink or InfiniBand NDR) and careful load balancing to avoid hot experts that receive disproportionate token traffic.

The mechanism is straightforward. The production engineering challenges are not.

Memory sizing is not as simple as "671B divided by GPU count." The active parameter framing is useful for compute cost estimation but misleading for memory sizing. You still store all 671B parameters - EP distributes them across GPUs, it does not eliminate them.

Additionally, most production MoE architectures include shared experts that run on every token regardless of routing. DeepSeek-V3 has shared experts that function as a dense component sitting alongside the sparse routed component. Memory budgeting must account for both the routed experts on each GPU and the shared expert weights that every GPU carries.

EP and TP are almost always combined. For large MoE models, individual expert weights may not fit on a single GPU, or the TP degree needed to meet latency SLOs requires splitting within each expert. Production deployments combine EP to distribute experts across GPUs with TP within each expert group. Sizing requires deciding both the EP degree - how many GPU groups to distribute experts across - and the TP degree within each group. These interact: higher EP degree reduces per-GPU expert memory but increases all-to-all communication surface; higher TP degree reduces per-GPU expert compute time but adds all-reduce overhead within each expert group.

The expert load imbalance problem is the critical production constraint. EP throughput is bounded by the busiest expert, not the average. If your workload causes one expert to receive 3× the average token traffic - which happens regularly due to training distribution biases - the entire EP group runs at roughly one-third of theoretical throughput regardless of how idle the other GPUs are. This is not a theoretical concern: in production deployments, unmeasured expert utilization distributions are the most common source of EP underperformance. Before committing to an EP configuration, measure your actual expert activation distribution on your specific workload. Production EP deployments mitigate this through two mechanisms: a capacity factor at the router level that caps how many tokens any single expert can receive per batch - forcing overflow tokens to a backup expert rather than stalling the pipeline - and expert-aware load balancing in the serving framework that monitors utilization in real time and rebalances routing weights when hot spots emerge. Neither mechanism eliminates imbalance entirely, but together they keep it manageable. The practical takeaway: always set a capacity factor above 1.0 in your serving configuration, and monitor per-expert token counts as a production metric from day one.

Expert overflow causes silent quality degradation. Each expert has a fixed capacity - a maximum number of tokens it can process per forward pass. When routing sends more tokens to an expert than its capacity allows, the overflow tokens are either dropped or rerouted to a backup expert. This does not throw an error - the model silently produces degraded output for overflow tokens. Setting the expert capacity factor too low to save memory is a common mistake that produces quality regressions that are difficult to attribute. In production, always monitor expert overflow rates as a first-class metric alongside throughput and latency.

EP adds latency that does not compress well. The all-to-all communication at each MoE layer is a synchronization barrier - every token must wait for the slowest expert to finish before computation proceeds to the next layer. Unlike the TP all-reduce which can be partially overlapped with computation, the EP all-to-all is harder to hide. At large EP degrees on slower interconnects, this latency accumulates across MoE layers and adds meaningfully to TTFT. EP is primarily a throughput optimization - it makes large MoE models economically viable at scale - but it is not a latency optimization. For strict TTFT SLOs, EP degree should be kept as low as memory constraints allow.

5.5 How Production Systems Combine These Strategies

The four strategies are building blocks. Production systems assemble them based on three independent constraints: does the model fit on one GPU, does it fit within one node, and is it a dense or MoE architecture? Each combination below addresses a specific combination of those constraints. No large-scale production deployment uses a single parallelism strategy in isolation. The standard patterns:

Combination	When Used	What Each Strategy Does	Key Constraint
TP × DP	Standard production setup (e.g., 70B-class models)	**TP** fits the model within a node and reduces latency **DP** replicates across nodes to scale throughput (RPS)	Memory and latency handled by TP. Capacity scaling via DP.
TP × PP	Model too large to fit within a single node even with maximum TP	**TP** distributes compute within a node **PP** splits layers across nodes	Inter-node bandwidth (InfiniBand / RoCE) becomes the bottleneck
TP × DP × EP	Large-scale MoE serving	**TP** handles attention layers within each node. **EP** distributes experts across GPUs. **DP** scales replicas for throughput	All-to-all communication for expert routing plus overall system load balance.

The decision logic is simpler than it looks. Start with a single GPU. If the model does not fit, apply TP within the node. If it still does not fit, add PP across nodes. Once the per-instance configuration is right, scale horizontally with DP. If you are serving MoE, layer EP on top of TP for the expert layers. You are almost never choosing between these strategies - you are stacking them in the right order for your specific constraints.

5.6 The Interconnect Is Not a Detail

Every combination in the previous section has one variable that determines whether the theoretical performance is actually achievable: the interconnect fabric between GPUs.

The interconnect landscape - which technology serves which transfer path

Different parallelism strategies generate fundamentally different data movement patterns, and each pattern has its own physical interconnect. Understanding which technology applies where is the prerequisite for making any inter-node procurement decision.

Transfer path	What moves	Technology	Typical bandwidth
GPU ↔ GPU (same node)	Activations, all-reduce gradients (TP)	NVLink (NVSwitch on DGX/HGX)	~900 GB/s bidirectional
GPU ↔ CPU DRAM (same node)	KV cache swap, model checkpoint	PCIe Gen4/Gen5	64-128 GB/s
GPU ↔ NVMe (same node)	Model weight loading at cold start	PCIe Gen4/Gen5 via NVMe controller	~12.5 GB/s
Node ↔ Node (inter-node)	PP activations, EP all-to-all, KV transfer in disaggregation	InfiniBand NDR or RoCE v2	25-50 GB/s per link
Node ↔ Object storage	Model artifact loading (S3, GCS)	Ethernet (10/25/100 GbE)	1-12 GB/s effective

The table makes the scoping explicit: NVLink solves intra-node TP. InfiniBand or RoCE solve inter-node PP and EP. PCIe solves CPU-GPU and NVMe transfers. These are not competing technologies - they are complementary layers in the same stack, each sized for a different bottleneck.

Intra-node: NVLink vs PCIe

The interconnect is the most common reason TP deployments underperform their theoretical speedup. Teams that benchmark TP=4 on PCIe and see disappointing latency frequently conclude that their model does not parallelize well - when the actual problem is that roughly 25% of every step is spent on communication rather than compute. NVLink does not just improve TP performance. It determines whether TP above 2 is worth deploying at all.

Within a DGX or HGX H100 node, NVLink provides ~900 GB/s of GPU-to-GPU bandwidth. PCIe Gen5 provides ~128 GB/s per x16 link - a 7× difference that is not marginal. At TP=2, PCIe overhead sits around 8% of step time - uncomfortable but manageable. At TP=4, it reaches ~25%. At TP=8, it exceeds 38% - more than a third of every step spent moving data rather than computing. The latency benefit of higher parallelism is progressively eroded until at TP=8 on PCIe the communication overhead effectively negates the compute saving.

The practical decision rule: TP=2 is acceptable on PCIe when NVLink is unavailable. TP=4 on PCIe should be benchmarked carefully against TP=2 before committing - the theoretical 2× compute benefit rarely survives the communication overhead in practice. TP=8 on PCIe is almost never worth deploying except for the most compute-bound prefill workloads where the arithmetic intensity is high enough to tolerate the communication tax.

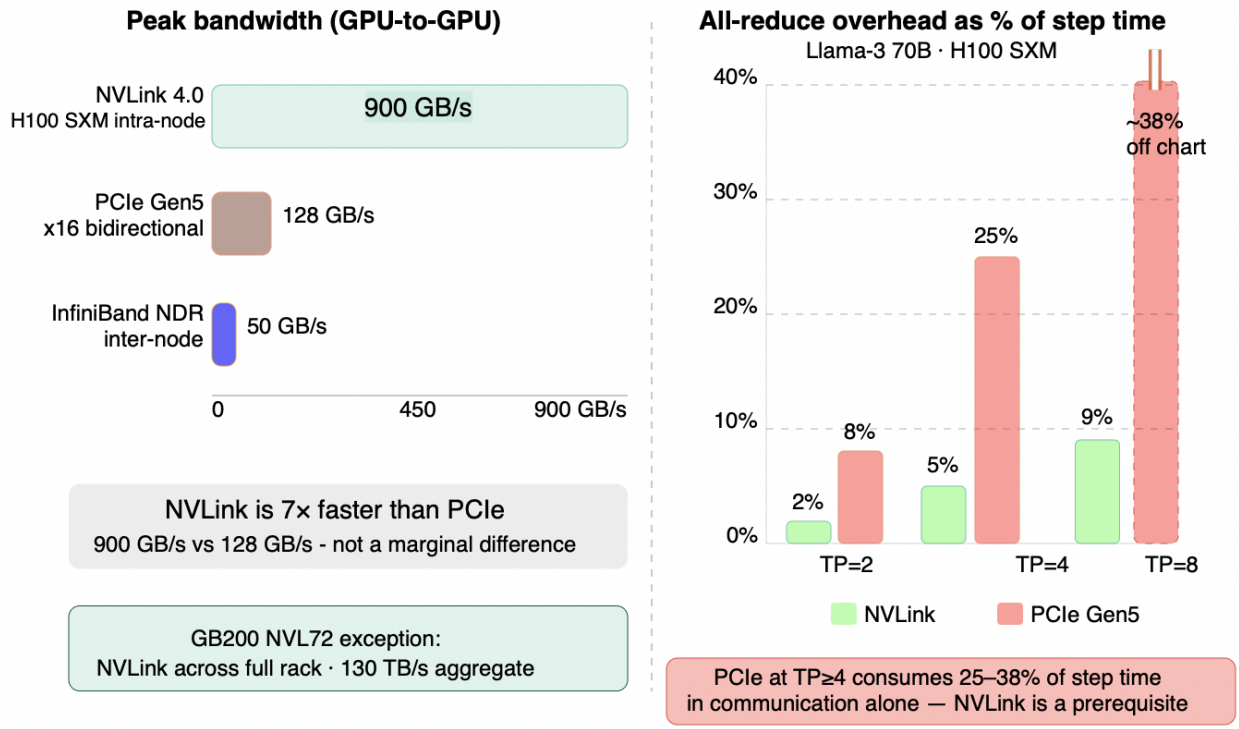

Figure 5.6 - NVLink vs PCIe: interconnect bandwidth and all-reduce overhead
Left: NVLink's 900 GB/s intra-node bandwidth is 7× faster than PCIe Gen5's 128 GB/s - not a marginal difference. InfiniBand NDR at ~50 GB/s governs inter-node communication for PP. Right: the consequence at different TP degrees. On NVLink, all-reduce overhead stays at 2-9% of step time across TP=2 through TP=8 - acceptable at any TP degree. On PCIe, overhead is already 8% at TP=2 and reaches ~25% at TP=4 - a quarter of every step spent on communication rather than compute. At TP=8 on PCIe the overhead exceeds 38%, effectively negating the latency benefit of higher parallelism. NVLink is a prerequisite for deployments with TP > 2.

Inter-node: InfiniBand NDR vs RoCE v2
For multi-node PP and EP deployments, inter-node bandwidth determines whether activation transfers and all-to-all communication remain manageable or dominate step time.

InfiniBand NDR (~50 GB/s per link) is purpose-built for GPU-to-GPU RDMA with hardware-managed congestion control. It works reliably out of the box for PP and EP traffic patterns at a significant cost premium over Ethernet.

RoCE v2 delivers comparable bandwidth over standard 100/400 GbE infrastructure at lower cost, but requires correct network configuration to behave reliably under load. When misconfigured, tail latency spikes under congestion and PP bubble fractions grow - eroding the parallelism gains the interconnect was supposed to enable.

Decision rule: use InfiniBand NDR for new multi-node deployments where reliability matters and budget allows. Use RoCE v2 where existing GbE infrastructure is in place and your networking team has RDMA experience. Either way, validate inter-node bandwidth under realistic PP or EP load before committing to an architecture that depends on it.

With the parallelism configuration and interconnect fabric determined, the unit of compute is defined. The next question is how to saturate it - which is the batching strategy covered in Section 6.

6 - Batching Strategy: From Static to Continuous to Disaggregated

Parallelism determines how many GPUs serve a model. Batching determines how efficiently those GPUs are used. A perfectly parallelized deployment running at batch=1 wastes most of its hardware - the roofline analysis showed exactly why: decode is memory-bandwidth-bound, and arithmetic intensity scales linearly with batch size. This section covers the scheduling strategies that close the gap between theoretical and realized throughput.

The stakes are concrete. A GPU holding a 70B model has the same VRAM whether it serves 1 concurrent request or 50. The difference in revenue per GPU-hour between those two operating points is entirely a function of batching strategy and scheduling policy - not hardware. Each evolution in batching covered in this section represents a discrete, measurable throughput improvement deployable on your existing fleet.

This section traces that evolution in order. Static batching fails structurally and the reasons are worth understanding precisely. The scheduler is the mechanism that makes everything else possible. Continuous batching is the production baseline that every serious deployment should already be running. Chunked prefill refines it further. Disaggregated prefill-decode is the architectural leap that unlocks the largest gains. Speculative decoding attacks a different bottleneck entirely - it does not change how requests are batched but how individual decode steps are executed, which is why it appears here as a complement to the batching strategies rather than a replacement for them.

6.1 Static Batching: Why It Fails

Static batching is the naive approach: collect N requests, pad all sequences to the length of the longest one, process the batch as a fixed group, return all results, then start the next batch.

The failure has two distinct mechanisms that are worth separating because they require different solutions.

The first is padding waste. All sequences in a static batch must be the same length for the GPU to process them as a single tensor operation. Every sequence shorter than the longest one is padded with dummy tokens to match. The GPU performs real computation on those padding tokens - consuming memory bandwidth and compute cycles - and produces output that is immediately discarded. At high input length variance, a substantial fraction of every batch's compute budget is spent on padding that contributes nothing to useful output.

The second failure is tail latency blocking. LLM output lengths vary dramatically per request - a simple factual question might complete in 20 tokens while a code generation request runs to 800. In a static batch, the GPU cannot start the next batch until every request in the current batch has finished generating its last token. A single long-running request holds the

entire batch hostage. Shorter requests that finished 10 seconds ago sit with completed results while the GPU continues generating tokens for the one slow request. During that wait, the GPU is producing output for exactly one request - the utilization equivalent of batch=1 - while the memory footprint of the entire batch is still occupied.

At high output length variance - which is the norm in any production LLM deployment - GPU utilization collapses for this reason. Static batching is not a suboptimal scheduling policy that can be tuned. It is architecturally incapable of handling variable-length autoregressive generation efficiently. The padding and tail latency problems are structural, not implementation deficiencies.

The solution requires a fundamentally different scheduling model - one that treats the active request set as a dynamic, continuously changing collection rather than a fixed group processed start-to-finish. That is what the continuous batching scheduler does. But before getting to continuous batching, it is worth understanding the scheduler itself as a standalone component, because the scheduler is what makes every subsequent optimization in this section possible.

6.2 The Scheduler: A First-Class Component

Before discussing any batching strategy, it is worth being precise about what a scheduler actually controls - because every batching strategy in this section is, at its core, a scheduling policy decision.

The LLM inference scheduler operates at the iteration level. On each forward pass, it decides three things: which requests are included in the current batch, how many tokens from each request are processed in this step, and what priority ordering governs admission when the batch is full. These decisions directly determine TTFT, ITL, queue depth, and GPU utilization - simultaneously and in tension with each other.

The complication that makes LLM scheduling harder than a simple queue is that requests have two distinct phases - a compute-intensive prefill and a memory-bandwidth-bound decode - and the scheduler must balance both simultaneously on the same hardware. A decision that improves TTFT for new arrivals directly degrades ITL for requests already in decode. Every scheduling policy is a choice about whose experience to protect. Understanding the scheduling policy families is the prerequisite for every batching decision that follows.

The four scheduling policy families - and what they cost you

Think of the GPU batch as a shared highway lane. The scheduler decides who gets to enter, in what order, and whether anyone can be overtaken. The choice of policy determines whose experience degrades when the lane gets congested.

FCFS (First-Come-First-Served) is the default in vLLM and most production frameworks - a strict queue in arrival order, regardless of prompt length or latency deadline. Simple to implement, blind to consequences. The problem is the slow truck on the highway: a single 64K-token request ahead of ten short chat requests blocks all of them, inflating their P99

latency. Input length variance in production is heavy-tailed, so this failure mode is not rare - it is the norm under mixed workloads.

Prefill-priority scheduling lets new arrivals cut to the front of the queue, immediately running their prefill step ahead of ongoing decodes. New users get their first token fast - TTFT improves. Every user already mid-response experiences a stutter - their ITL spikes by exactly the duration of the incoming prefill. At high arrival rates with long prompts, this produces continuous jitter that makes streaming feel broken even when average latency is within budget. The tradeoff is mathematically exact: prioritizing prefill reduces TTFT and unavoidably worsens time-between-tokens in a colocated system.

Decode-priority scheduling is the opposite: protect active responses from any interruption. ITL is stable and predictable. New requests queue and wait - TTFT grows linearly with queue depth. TensorRT-LLM and FasterTransformer use this approach, fully completing current batches before admitting new ones. Under high load, new users wait seconds before seeing their first token while the GPU serves everyone already in flight.

SLO-aware priority scheduling is the production answer to all three failure modes. Instead of ordering by arrival time or phase type, it tracks each request's remaining latency budget. A request close to violating its TTFT deadline moves to the front. A request with 800ms of budget remaining yields to one with 50ms left - regardless of who arrived first. This is what makes interactive chat, batch summarization, and RAG pipelines coexist on a shared fleet without one class consistently cannibalizing another's SLO.

SLO-aware scheduling requires estimating each request's remaining generation time - which is non-trivial since output length is not known at admission in autoregressive generation. Production implementations use either output length predictors trained on historical workload distributions, or conservative deadline tracking based on worst-case output length assumptions. The prediction accuracy directly affects scheduling quality - a poor output length estimator degrades SLO-aware scheduling toward FCFS behavior in practice.

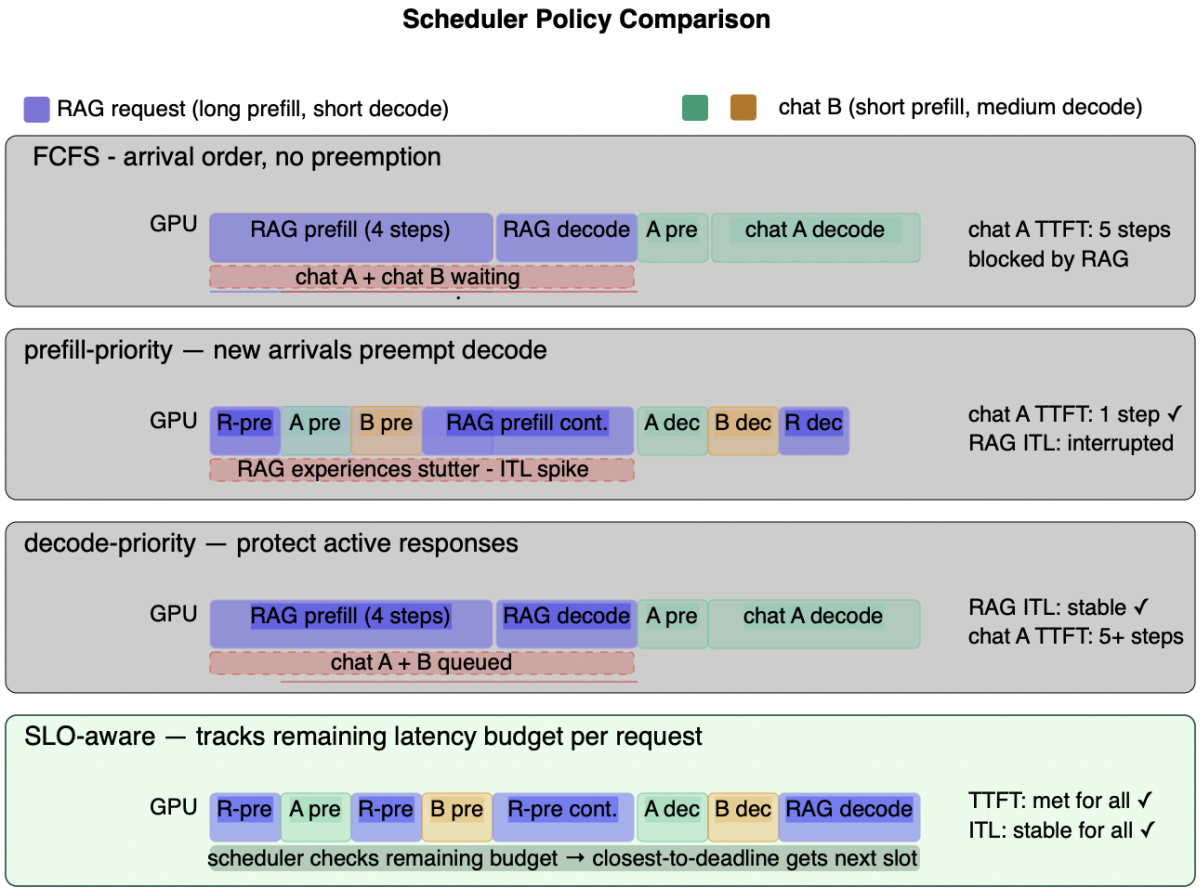

Figure 6.2 - The four scheduler policies: same three requests, four different outcomes
Three requests arrive in sequence: a RAG query with a long prefill, and two short chat requests. Each row shows how the GPU is allocated under a different scheduling policy. FCFS (top) processes requests in strict arrival order - the long RAG prefill blocks both chat requests, inflating their TTFT by 5 steps with no awareness that they had tighter latency budgets. Prefill-priority (second) admits chat A and B immediately, giving both fast TTFT - but RAG's decode is interrupted each time a new prefill arrives, causing ITL spikes that make streaming feel broken. Decode-priority (third) protects active responses from any interruption - RAG's decode is smooth, but chat A and B wait 5+ steps before seeing their first token, with TTFT growing linearly with queue depth. SLO-aware (bottom) tracks each request's remaining latency budget and routes the closest-to-deadline request first - all three requests meet their TTFT and ITL targets simultaneously. The summary table shows why FCFS and single-phase policies consistently fail mixed production workloads.

Why the scheduler matters before choosing a batching strategy. The scheduler determines which requests are in the batch when a prefill arrives, how aggressively prefill is chunked, and how decode requests are prioritized in disaggregated serving. A sophisticated batching architecture running on FCFS will consistently underperform a simpler architecture with an SLO-aware scheduler - the scheduling policy sets the ceiling on what any batching optimization can achieve.

6.3 Continuous Batching: The Baseline

The scheduler policies govern how requests compete for the batch. Continuous batching changes the more fundamental question underneath that competition: when does the batch itself turn over? Before continuous batching existed, the serving system made one scheduling decision per batch: assemble N requests, run them all to completion, then accept new ones. The GPU idled whenever any request finished early - and with variable output lengths, that happened on nearly every batch.

The fix - first formalized in the Orca research paper (OSDI 2022) and now implemented in every serious framework under different names - is iteration-level scheduling. After every single forward pass, completed requests are evicted and new ones are admitted immediately. The GPU stays continuously occupied rather than waiting for the slowest request in each group to finish. vLLM calls it continuous batching. TensorRT-LLM calls it in-flight batching. SGLang and TGI use the same mechanism under their own implementations. The concept is identical across all of them.

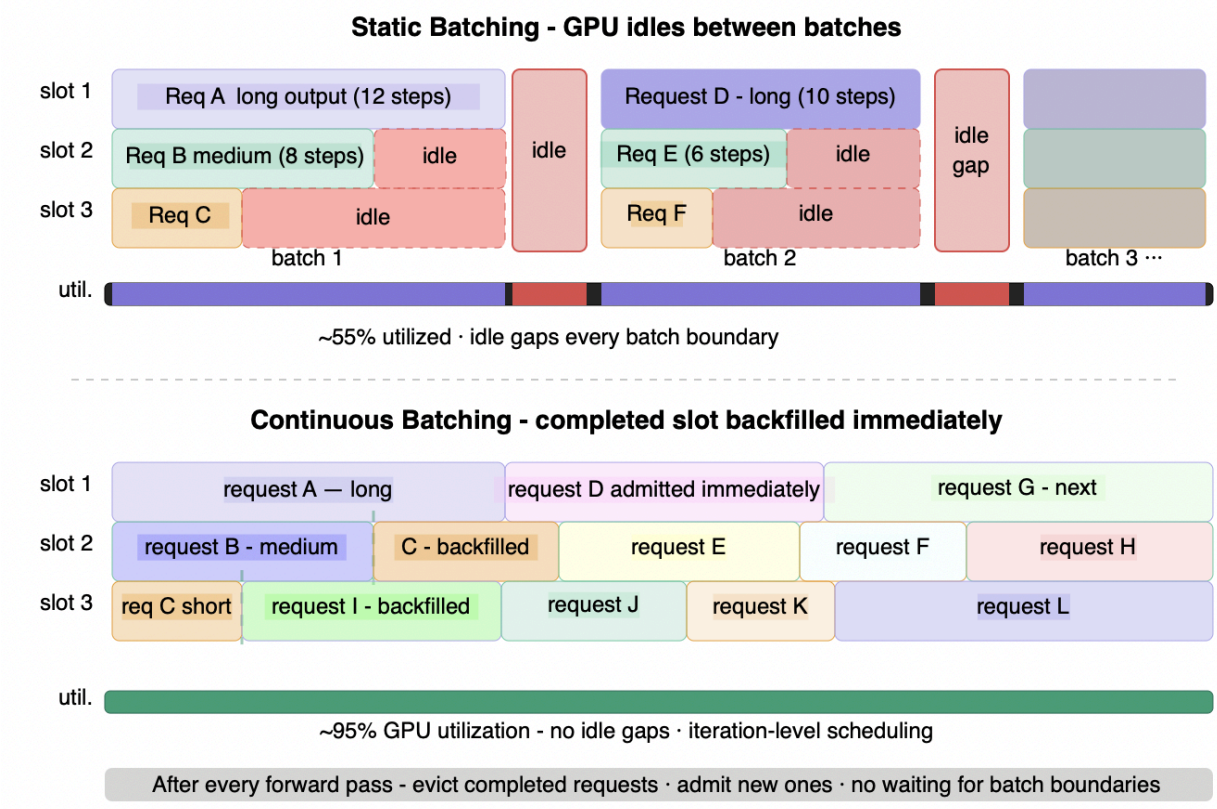

Figure 6.3 - Static batching vs continuous batching: GPU slot utilization
Static batching (top) runs each batch to full completion before admitting new requests. Shorter requests finish early and their slots sit idle - waiting for the slowest request. The inter-batch gap makes it worse: the GPU goes fully dark at every batch boundary. Continuous batching (bottom) makes one scheduling decision per forward pass. The moment any slot completes, a new request fills it - no batch boundaries, no idle gaps, no waiting. The utilization improvement is structural, not incremental: from ~55% to ~95% on the same hardware, serving the same requests.

This is table stakes. If your serving stack does not support it, nothing else in this section will help. But enabling continuous batching is not the end of the story - it introduces three production challenges that must be managed explicitly.

KV cache pressure is the primary operational risk. Continuous batching maximizes GPU slot utilization, which simultaneously maximizes concurrent KV cache consumption. At high throughput, the KV cache fills completely and new requests cannot be admitted even though the GPU has compute headroom. The system hits a memory wall before a compute wall. This is the most common failure mode for teams that enable continuous batching without tuning their KV cache budget. It is also why continuous batching and PagedAttention are inseparable in production - PagedAttention's dynamic memory allocation is what prevents KV cache fragmentation from artificially capping concurrency below what the raw memory budget would allow.

Preemption is the safety valve - and it has a cost. When the KV cache fills completely and a high-priority request must be admitted, the scheduler must evict an active request's KV cache from GPU memory. This happens either by swapping KV tensors to CPU DRAM - adding memory bandwidth overhead - or by discarding them entirely and recomputing from scratch when the request is resumed - adding compute overhead proportional to the evicted context length. Under sustained high load with aggressive admission, preemption frequency rises and P99 latency degrades significantly even when average latency looks healthy. Preemption events are a critical production metric to monitor - a sudden spike in preemption rate is an early warning that your KV cache budget is too small for your current traffic pattern.

Admission control determines P99 behavior under load. Continuous batching without admission control accepts every request immediately and preempts existing ones to accommodate them when necessary. Continuous batching with admission control queues new requests when the KV cache budget is exhausted rather than preempting. This is a configuration choice that most teams leave at framework defaults without understanding the tradeoff: queuing protects P99 latency for active requests at the cost of higher TTFT for new arrivals; preemption protects TTFT at the cost of latency spikes for requests that get evicted. The right choice depends on which SLO your workload weights more heavily - and it should be an explicit decision, not a default.

The utilization gain scales with output length variance. The ~55% to ~95% improvement in the figure above reflects a high output length variance workload - the typical production scenario for mixed chat, RAG, and code generation traffic. For workloads with low variance - batch document processing where every document produces roughly the same output length - continuous batching provides smaller gains because the tail latency blocking problem is smaller to begin with. If you enable continuous batching and see modest throughput improvement, measure your output length distribution before concluding the implementation is wrong. The workload profile may simply not benefit as strongly.

6.4 Chunked Prefill: Interleaving Without Full Separation

Continuous batching keeps the GPU continuously occupied. What it does not solve is the interference between a new request's prefill and active decode requests competing for the

same forward pass. A long prefill still stalls every decode in the batch for its full duration - 300-500ms on an H100 for a 16K-token prompt on a 70B model. Chunked prefill is the targeted fix for that specific problem, without requiring separate hardware.

The idea from SARATHI is straightforward: instead of processing the full prompt in one blocking step, split it into fixed-size chunks - say 256 tokens - and process one chunk per iteration alongside ongoing decode requests. No single iteration is monopolized by a long prompt. Decode requests continue generating tokens between every chunk.

This works because of the roofline insight from earlier sections applied directly. A prefill chunk is compute-bound - it saturates CUDA cores. Decode requests in the same batch are memory-bandwidth-bound - they saturate HBM. They consume different hardware resources simultaneously, which means batching them together in the same forward pass approaches both ceilings at once rather than alternating between them. GPU utilization improves as a direct consequence.

Think of it like a highway where a single articulated truck - a full 16K prefill - used to block all traffic for 300-500ms every time it entered the road. Chunked prefill breaks that truck into a convoy of smaller vehicles. Each segment takes its turn in the shared flow of traffic, and the motorcycles (decode steps) keep moving between each segment. The road is still shared - nobody gets their own lane. But no single vehicle monopolizes it long enough to cause a visible stall. That separation into dedicated lanes is what disaggregation in the next section actually does.

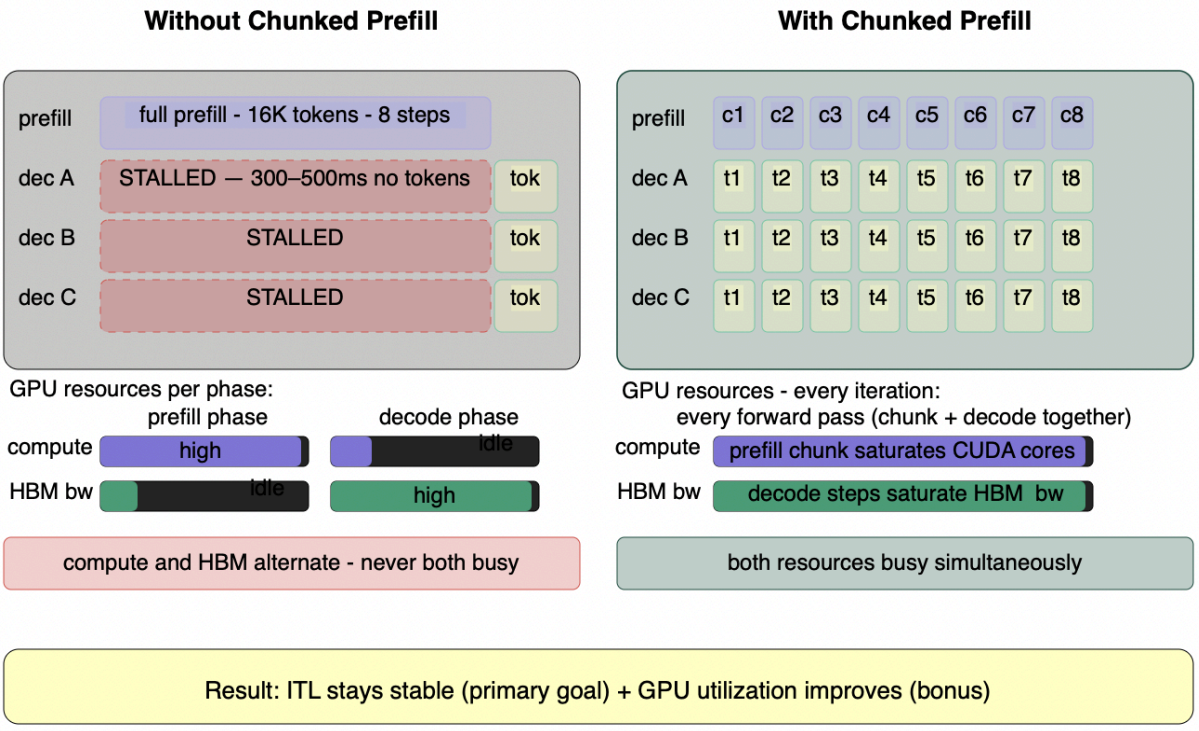

Figure 6.4 - Chunked prefill: ITL stability and GPU utilization
Without chunked prefill (left), the GPU alternates between two extremes - compute saturated and HBM idle during full prefill, then HBM saturated and compute idle during decode. Every active decode request stalls completely for 300-500ms while the long prefill runs. With chunked prefill (right), each forward pass contains both a 256-token prefill chunk and decode steps from all active requests. The prefill chunk saturates CUDA cores. The decode steps saturate HBM bandwidth. They use different hardware resources - no competition, no stall. Both ceilings are approached simultaneously in every iteration. The primary benefit is ITL stability: decode tokens flow continuously rather than stalling. The secondary benefit is GPU utilization: the two resources that previously alternated are now both busy at once.

The tradeoff is explicit and worth stating precisely. Chunked prefill improves ITL for active decode requests at the cost of higher TTFT for the request being chunked. A 16K prompt split into 64 chunks of 256 tokens takes 64 iterations to complete prefill rather than 1 - TTFT for that request increases proportionally. Chunk size is therefore a tunable parameter that balances two competing objectives: larger chunks complete prefill faster, reducing TTFT, but cause longer decode stalls per iteration; smaller chunks minimize stall duration but extend TTFT for the chunked request. Most production deployments settle on 512 or 1024 tokens as the chunk size - large enough to maintain GPU tile utilization, small enough to keep per-iteration decode interference below the perceptible threshold.

One constraint that cannot be ignored: chunk size must align to GPU matrix multiplication tile dimensions. GPUs execute matrix multiplications in fixed-size tiles - typically multiples of 128 tokens on current hardware. Chunk sizes that do not align to these boundaries cause partial tile utilization, reducing the compute efficiency that makes chunked prefill worthwhile. Always set chunk size to a power of 2 and validate utilization metrics after tuning.

In vLLM's V1 engine, chunked prefill is enabled by default. SGLang supports it natively. No configuration beyond chunk size tuning is required in either framework.

Chunked prefill is the right first intervention for ITL instability - it costs nothing, ships in your existing framework, and eliminates the worst decode stalls at mixed workloads. What it cannot do is give prefill and decode their respective optimal hardware environments. For workloads where that matters - high-throughput serving at scale where the roofline mismatch between prefill and decode represents meaningful wasted GPU capacity - disaggregation is the architectural step that chunked prefill is pointing toward.

6.5 Disaggregated Prefill-Decode: The Architectural Solution

Chunked prefill reduces interference within a colocated system by time-slicing the prefill across iterations. Disaggregated prefill-decode eliminates interference architecturally by assigning each phase to dedicated GPU pools that never share execution slots.

Prefill instances receive incoming requests, execute the full prefill pass, and populate the KV cache. That KV cache is transferred to a decode instance via RDMA (Remote Direct Memory Access - a protocol that transfers data directly between GPU memory pools across nodes without CPU involvement) - InfiniBand NDR, AWS EFA, or NVLink for intra-node transfers - which then handles all token generation independently. The scheduler at each pool operates without interference from the other: the prefill scheduler optimizes for prompt

processing throughput; the decode scheduler optimizes for stable ITL and decode concurrency.

DistServe (OSDI 2024) formalized this using queuing theory - modeling prefill and decode as two independent queues, each with its own arrival rate, service time, and resource pool, and showing that optimizing each independently produces substantially better goodput than any colocated configuration where both phases compete for the same resources. By late 2025, every major production-grade serving framework had first-class support for disaggregation - SGLang and NVIDIA Dynamo treat it as a core serving pattern, Ray Serve LLM and llm-d ship it as a documented deployment mode, and vLLM continues hardening it toward a stable release.

The gains from combining disaggregation with large-scale expert parallelism on MoE models are concrete: SGLang on 96 H100 GPUs achieves 52.3K input tokens/sec and 22.3K output tokens/sec per node on DeepSeek-R1 - a 5× improvement over vanilla tensor parallelism on the same hardware.

Sizing the P:D ratio. Disaggregation is only as good as the ratio you set. Too many prefill GPUs and decode becomes the bottleneck; too few and incoming requests queue behind long prefills.

The intuition is simple: prefill cost scales with input length, decode cost scales with output length. A workload with 4K-token inputs and 200-token outputs is compute-heavy on the way in - it needs more prefill GPUs. A code generation workload with 100-token inputs and 3K-token outputs is the reverse - most of the work happens during decode.

To derive the ratio from your actual workload, benchmark each phase independently to get tokens/sec/GPU, then:

```
P:D ratio = (peak_RPS × avg_input_length / prefill_throughput) ÷
            (peak_RPS × avg_output_length / decode_throughput)
```

In a colocated system this ratio is permanently fixed at 1:1 regardless of what your workload actually demands. With disaggregation you set it explicitly - and re-tune it as your input/output mix evolves without reprovisioning the entire fleet. As a starting point: RAG pipelines with long retrieved contexts typically run P:D ratios of 1:3 to 1:4 - most of the wall-clock time is in decode. Code generation workloads with short inputs and long outputs often run 1:6 or higher. Chat workloads with balanced input/output lengths typically start at 1:2 and tune from there.

The KV cache transfer tax. Every disaggregated request pays a fixed latency cost to move the KV cache from the prefill node to the decode node. Before committing to this architecture, calculate whether that cost is worth it for your workload. The math is simple:

```
transfer time = KV cache size ÷ network bandwidth.
```

For a concrete example - Llama-3 70B at FP8 with a 4K-token prompt, the KV cache is roughly 320 MB. Over InfiniBand NDR that transfer takes about 0.8ms. If a long prefill was

previously stalling your decode batch for 400ms, paying 0.8ms to eliminate that stall is an easy decision.

Now flip it: a 200-token prompt generates only ~16 MB of KV cache - transfer time drops to 0.04ms, but the interference you are eliminating is also tiny. At short prompt lengths, running prefill locally on the decode node is often faster end-to-end than shipping the KV cache across the network. The practical implication: disaggregation is most valuable for workloads with long input contexts. For workloads dominated by short prompts - under 500 tokens - the interference being eliminated is small enough that chunked prefill on a colocated system is the better choice.

Scheduler considerations in disaggregated deployments. With separate prefill and decode pools, scheduling splits across two independent coordinators. The prefill scheduler manages admission, request ordering, and load balancing across prefill instances. The decode scheduler manages KV cache memory pressure, batch composition, and per-request SLO tracking. A KV-aware router connects the two - directing incoming requests to prefill instances and routing completed KV transfers to decode instances with available capacity.

Llm-d (a Kubernetes-native LLM serving infrastructure project) and NVIDIA Dynamo (NVIDIA's production disaggregated serving runtime) both implement cache-aware routing: where possible, requests are sent to decode instances that already hold relevant prefix cache entries, avoiding redundant KV transfers for workloads with shared system prompts or repeated context.

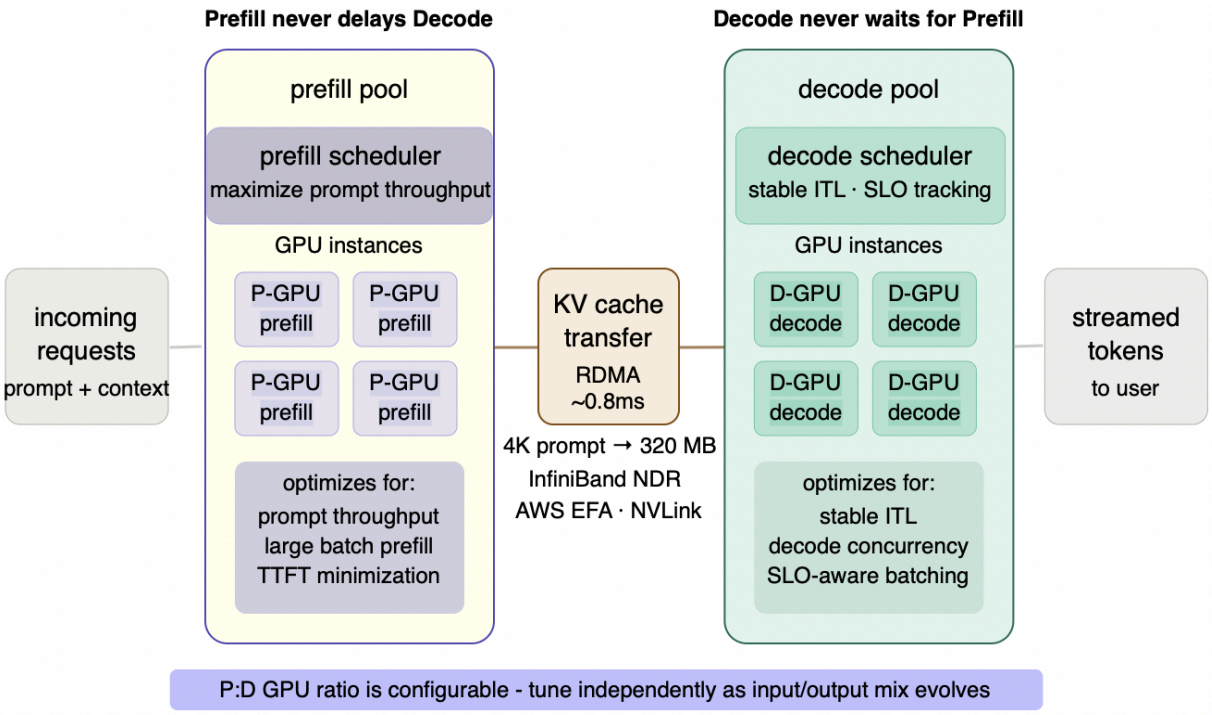

Figure 6.5 - Disaggregated prefill-decode architecture
Incoming requests flow into the prefill pool, which executes the full prompt processing pass and transfers the resulting KV cache via RDMA to the decode pool. From that point, the decode pool

handles all token generation independently - no prefill activity ever touches the decode execution path again. Each pool has its own scheduler optimizing for different objectives: the prefill scheduler maximizes prompt throughput and minimizes TTFT; the decode scheduler maintains stable ITL and tracks per-request SLOs. In a colocated system these objectives conflict on every forward pass. Disaggregation makes them independent. The P:D ratio - how many GPUs each pool gets - is configurable and re-tunable as your workload mix evolves. A colocated deployment is permanently fixed at 1:1 regardless of whether your workload is prefill-heavy or decode-heavy.

6.6 Speculative Decoding: Breaking the Decode Bandwidth Ceiling

Every batching strategy in this section - continuous batching, chunked prefill, disaggregation - improves how requests are scheduled and phased. None of them address a more fundamental constraint: during decode, the GPU generates exactly one token per forward pass, loading the full model weights from HBM on every step. At small batch sizes, CUDA cores sit largely idle while the memory subsystem does the work. The roofline model from Section 3 makes this explicit - decode is memory-bandwidth-bound, not compute-bound, and no scheduling optimization changes that.

Speculative decoding attacks this constraint directly by changing what gets fed into the target model's decode step.

Step 1 - Draft model speculates ahead. A small fast draft model (typically 1-3B parameters) runs 4-8 cheap decode steps autoregressively, generating candidate tokens `[t1, t2, t3, t4, t5]` along with their probability distributions `q(t1..t5)` from its own weights and KV cache.

Step 2 - Target model verifies in one parallel pass. Normally, the target model's prefill stage produces one token which is fed back into decode as the next input - one token in, one token out, repeat. Speculative decoding replaces that single token input with the draft model's 5 candidate tokens fed in together. The target model processes all 5 simultaneously - extending its existing KV cache with keys and values for all 5 positions in one parallel pass - and produces its own probability distributions `p(t1..t5)` for each position. Because multiple token positions are processed in parallel, this pass runs compute-bound rather than memory-bandwidth-bound - exactly the roofline shift described in Section 3.

Step 3 - Token acceptance via probability comparison. For each position, the target model compares its distribution against the draft model's. Where they agree, the token is accepted. Where they first diverge, that position and everything after it is rejected and the KV cache rolls back. The target model samples a corrected token from its own distribution at that position - using probabilities already computed during the verification pass, at no extra cost. Worst case - every draft token rejected - you still get one corrected token from one forward pass, identical to vanilla decode. You never do worse than baseline.

The gain is not that the forward pass gets cheaper - the target model still loads all its weights from HBM on every verification pass. The gain is in **tokens produced per forward pass**. Vanilla decode: one weight-loading trip → one token out. Speculative decoding: one weight-loading trip → three or four tokens out on average. The memory bandwidth cost is amortized across multiple accepted tokens. Think of it as a delivery truck making the same warehouse trip but dropping packages at four houses instead of one - same trip cost, four

times the useful work. The output distribution is mathematically identical to what the target model would have produced alone - speculative decoding is lossless, not an approximation.

What this means on the roofline. Speculative decoding shifts decode from the memory-bandwidth-bound regime toward the compute-bound regime by batching multiple token verifications into a single pass. GPU compute utilization during decode - which typically sits at 20-30% for single-token autoregressive generation - rises meaningfully with speculation. With 3-4 tokens accepted per verification pass, the effective arithmetic intensity increases by the same factor, pushing utilization from the 20-30% range toward 60-80% at typical acceptance rates. This is not a full escape from the memory-bandwidth-bound regime but a meaningful shift up the roofline slope.

Acceptance rate is the critical variable. If the draft model's token distribution closely matches the target model's, acceptance rates of 70-90% per token are achievable, yielding 2-4× speedups on latency-sensitive workloads. If the draft model is poorly matched to the domain or the target model's distribution, acceptance rates drop and speculative decoding becomes slower than vanilla autoregressive generation - you pay the overhead of running the draft model without the benefit. Always measure acceptance rate in production; it is the metric that tells you whether speculative decoding is helping or hurting.

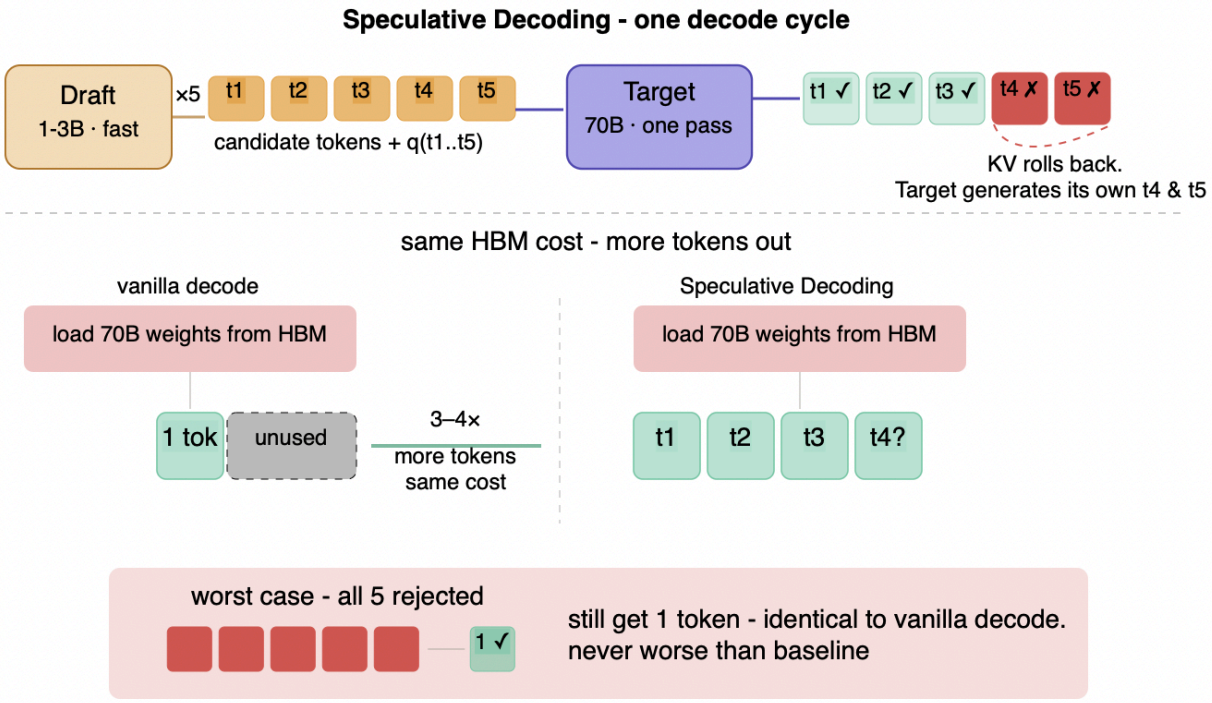

Figure 6.6 - Speculative decoding: mechanism and HBM amortization
Top: the draft model runs 5 cheap autoregressive steps producing candidate tokens. The target model verifies all 5 in one parallel pass - t1, t2, t3 accepted; t4 diverges triggering a rollback to t3 and sampling a corrected t4 from the target's own distribution. Result: 3 tokens from one target model forward pass. Middle: the gain is purely amortization - the same HBM cost of loading all model weights produces 1 token in vanilla decode and 3-4 tokens in speculative decoding. Bottom: worst case guarantees the floor - even if all draft tokens are rejected, you still get 1 corrected token,

identical to vanilla decode. The output distribution is mathematically identical to the target model alone.

Three deployment patterns in production:

- **Separate draft + target models** - a small model (1-3B parameters) runs alongside the large model. Works best when a purpose-trained draft model exists for the target model family. Llama-3 8B as draft for Llama-3 70B is a common pairing. One memory implication worth calculating before deployment: a 3B draft model at FP16 adds 6 GB alongside the target model - modest relative to a 70B target but not free. In memory-constrained configurations, prefer N-gram speculation or EAGLE draft heads instead.
- **N-gram speculation** - uses n-gram matching on the prompt and previous outputs to generate draft tokens without a separate model. Zero additional memory cost, effective for repetitive or structured outputs (code, templates, JSON). Supported natively in vLLM and SGLang.
- **EAGLE / Medusa (draft heads)** - adds lightweight prediction heads to the target model itself, trained to speculate future tokens. Higher acceptance rates than separate draft models for matched domains; requires fine-tuning the draft heads on your target distribution.

When speculative decoding helps and when it does not:

Scenario	Verdict	Reason
Low-to-medium concurrency, latency-sensitive	High impact	Memory bandwidth is the clear bottleneck; speculation raises arithmetic intensity
High concurrency, throughput-optimized	Low or negative impact	At large batch sizes decode becomes more compute-bound; speculation overhead hurts
Repetitive or structured output (code, JSON)	High impact	N-gram speculation achieves high acceptance rates at near-zero overhead
Creative or highly variable output	Lower impact	Draft model acceptance rates drop; measure before committing
Mixed concurrency with variable output length	Measure first	Acceptance rate variance across request types makes aggregate impact unpredictable - benchmark on your actual workload distribution before committing
Disaggregated P/D deployment	Compatible	Speculation runs on decode instances independently; prefill instances unaffected

The strategies in this section are not mutually exclusive - continuous batching, chunked prefill, disaggregation, and speculative decoding compose together. The next section

provides the decision framework for which combination makes sense for your specific workload profile.

6.7 Choosing the Right Strategy

The strategies in this section are additive, not mutually exclusive. Continuous batching is always the foundation. Chunked prefill and SLO-aware scheduling layer on top for mixed workloads. Disaggregation is the architectural escalation for large fleets with strict, independent latency targets.

Scenario	Strategy	When to move up
Single GPU or small fleet	Continuous batching, FCFS scheduler	P99 ITL spikes appear under mixed input lengths - short requests stalling behind long prompts
Mixed workload, variable prompt lengths	+ Chunked prefill, SLO-aware scheduler	TTFT and ITL SLOs are independently defined and both need to be met simultaneously
Low-to-medium concurrency, latency-critical	+ Speculative decoding	Acceptance rate stays above 60% on your workload; memory bandwidth is still the decode bottleneck
Large fleet, strict independent SLOs for TTFT and ITL	+ Full P/D disaggregation, KV-aware routing	Colocated chunked prefill cannot meet both TTFT and ITL targets simultaneously at peak load
Offline batch only	Maximum batch size, FCFS acceptable	N/A - throughput is the only metric; latency SLOs do not apply

7 - KV Cache Optimization: Your Most Under-Used Lever

Two systems running the same model on identical hardware can differ significantly in effective throughput based solely on KV cache strategy. This is where naive deployments diverge most from optimized production systems - and where the highest-leverage gains require no hardware changes.

7.1 PagedAttention: The Foundation

Before PagedAttention, serving systems allocated a single contiguous block of GPU memory per request at admission time, sized for the maximum possible output length. The problem is fundamental: output length is unknown when a request arrives. If the system's maximum context is 8K tokens and a request generates only 200 output tokens, 97% of its reserved memory sits locked and empty for the entire request lifetime - not reusable by any other request until that request fully completes. The vLLM paper reports that earlier systems utilized only 20-38% of allocated KV cache memory as a result. Not because the GPU lacked memory, but because the allocator held it hostage in large, mostly-empty reservations.

Think of it like a restaurant that assigns an eight-seat table to every party regardless of size. A couple gets a table for eight, six seats sit empty all evening, and the next party waits at the door even though the room is technically "full." The seats exist - they are just not usable.

PagedAttention fixes this by borrowing the virtual memory paging concept from operating systems. Instead of one contiguous slab per request, the KV cache is divided into small fixed-size blocks - 16 or 32 tokens per block in vLLM. A block table maps each request's logical token positions to physical GPU memory blocks, exactly like a page table maps virtual addresses to physical memory pages. Blocks from different requests are freely interleaved in physical memory. As a sequence grows it gets the next available free block regardless of physical location. When a request completes - or even when individual blocks are no longer needed - those blocks are immediately returned to the free pool at block granularity, not request granularity. That finer release cadence is the mechanism behind the 2-4× throughput improvement the vLLM paper reports on the same hardware.

Block size is a configuration tradeoff worth understanding. Smaller blocks - 16 tokens - reduce internal fragmentation: a request generating 17 tokens wastes only 15 tokens of padding rather than a full slab. But smaller blocks mean larger block tables and higher metadata overhead per request. Larger blocks - 128 tokens - reduce metadata overhead but increase fragmentation for short requests. The default of 16 tokens is right for mixed workloads with variable output lengths. Increase block size only for workloads with

consistently long outputs where fragmentation is negligible and metadata overhead becomes the binding cost.

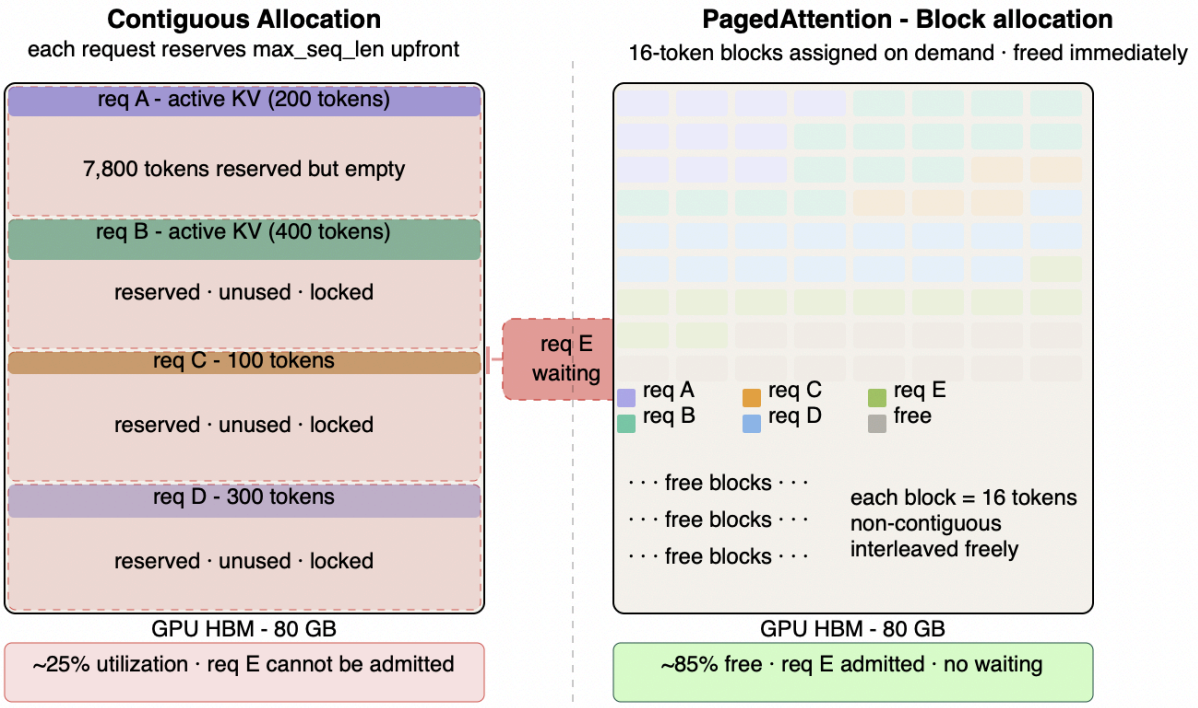

Figure 7.1 - PagedAttention: contiguous allocation vs block allocation
Left: each request reserves a full max_seq_len slab at admission. Four requests with short actual outputs hold 75%+ of GPU memory in empty reservations. Request E cannot be admitted despite most of the memory being physically free - it is just locked inside other requests' slabs. Utilization collapses to 20-38% under real workloads. Right: the same four requests occupy only the blocks they actually need - 16 tokens per block, non-contiguous, freely interleaved in physical memory. Request E is admitted immediately. Blocks are returned to the free pool the moment they are no longer needed, not when the entire request completes. The same GPU serves significantly more concurrent requests at near-full utilization.

One important constraint in distributed deployments. KV cache blocks are local to each GPU - there is no cross-GPU block pool. In a TP=4 deployment each GPU manages its own block allocator independently. This means the KV cache bottleneck in a TP group is always the most memory-pressured GPU. If GPU 0 exhausts its block pool, the request stalls regardless of how many free blocks GPU 1, 2, or 3 still have. PagedAttention eliminates within-GPU fragmentation efficiently - it does not solve cross-GPU memory imbalance. This matters in practice for TP deployments serving long-context workloads where attention head sharding creates uneven KV pressure across GPUs.

PagedAttention also enables prefix sharing across requests: blocks covering identical prompt prefixes - a shared system prompt, a few-shot template - can be physically shared in memory rather than duplicated per request. This is the foundation that prefix caching builds on in Section 7.2.

PagedAttention is now the default KV cache management mechanism in vLLM, TensorRT-LLM, SGLang, and every serious production inference framework. It is not an optimization you enable - it is the baseline everything else in this section builds on top of.

7.2 Prefix Caching: The Most Under-Deployed Optimization

Many production workloads share large common prefixes across requests - system prompts, RAG retrieved context, few-shot examples, conversation history. Without prefix caching, the KV state for these tokens is recomputed on every single request, burning GPU compute and memory bandwidth identically on the hundredth request as on the first. It is the inference equivalent of recalculating the same spreadsheet from scratch every time you open it rather than saving the result.

Prefix caching stores the computed KV blocks for reusable token sequences and reuses them on cache hits. When an incoming request matches a cached prefix, those tokens skip the prefill computation entirely - their KV state is read directly from cache. The effective prefill cost drops to only the unique tokens in the request. For a RAG workload with a fixed 8K system prompt and 4K retrieved context, a cache hit means prefilling only the user's 50-token query rather than 12,050 tokens. TTFT drops proportionally. GPU compute is freed for other requests.

The cache hit rate is everything - and it is workload-dependent. Before deploying prefix caching, estimate your expected hit rate. The signal is in your request distribution:

If your workload has a fixed system prompt shared across all requests - a customer service bot, a RAG pipeline with a standard context template, a code assistant with a fixed few-shot preamble - your hit rate on that prefix approaches 100% regardless of query diversity. This is where prefix caching delivers its largest gains, often reducing effective TTFT by 60-80% on the shared portion.

If your workload has highly variable or unique prompts - creative writing, open-ended chat with no shared context - hit rates approach zero and prefix caching adds overhead without benefit. The cache lookup cost is small but not free.

Multi-turn conversation history is the middle case. The system prompt is always cached. Each turn extends the cached prefix by one exchange. Hit rates on the accumulated history depend on how consistently the same conversation lands on the same instance - which is a routing problem covered in Section 7.3.

Two implementation approaches with different tradeoffs. vLLM uses hash-based exact prefix matching - the entire prefix token sequence is hashed and looked up in the cache. This is simple and fast but only matches requests that share an identical prefix from the start. SGLang implements a radix tree structure that enables partial prefix matching - any shared prefix length produces a cache hit, not just full-prefix matches. For workloads where requests share a system prompt but diverge in the retrieved context, radix tree matching captures partial hits that hash-based matching misses entirely. If your framework supports it, radix tree prefix caching is strictly better for mixed workloads.

The memory tradeoff must be sized explicitly. Prefix caching consumes KV cache blocks to store cached prefixes. Those blocks are unavailable for active request KV state. At high cache utilization, the LRU eviction policy begins evicting cached prefixes to make room for active requests - which reduces hit rates precisely when the system is under the highest load. The practical sizing rule: allocate at least 20-30% of your KV cache budget to prefix storage beyond what active request concurrency requires. Monitor both cache hit rate and eviction rate as production metrics - a rising eviction rate under load signals that your prefix cache budget is too small for your traffic pattern.

In a single-instance deployment, prefix caching is straightforward - one cache, all requests hit it. In a multi-replica deployment behind a load balancer, the same request may land on different replicas with cold caches, eliminating the hit rate benefit entirely. That is the problem Section 7.3 addresses.

7.3 Cache-Aware Routing: Single-Instance vs Multi-Replica

Prefix caching only delivers value if requests land on replicas that already hold the relevant KV state. In a naive round-robin setup, requests are distributed evenly across replicas regardless of content - which dilutes cache locality and collapses hit rates to roughly 1/N where N is the replica count. A perfectly configured prefix cache on each replica produces near-zero aggregate benefit if routing sends each request to a different one.

Cache-aware routing restores locality by routing requests based on their prefix - so that repeated or similar inputs consistently land on the same replica. After the first request warms a replica's cache for a given prefix, every subsequent request for that prefix hits without recomputation. The 4K-token RAG context prefill is paid once and amortized across all subsequent requests. This is not a marginal improvement - it is the difference between a cache hit rate of ~25% under round-robin and ~90%+ under prefix-hash routing for workloads with high prefix overlap.

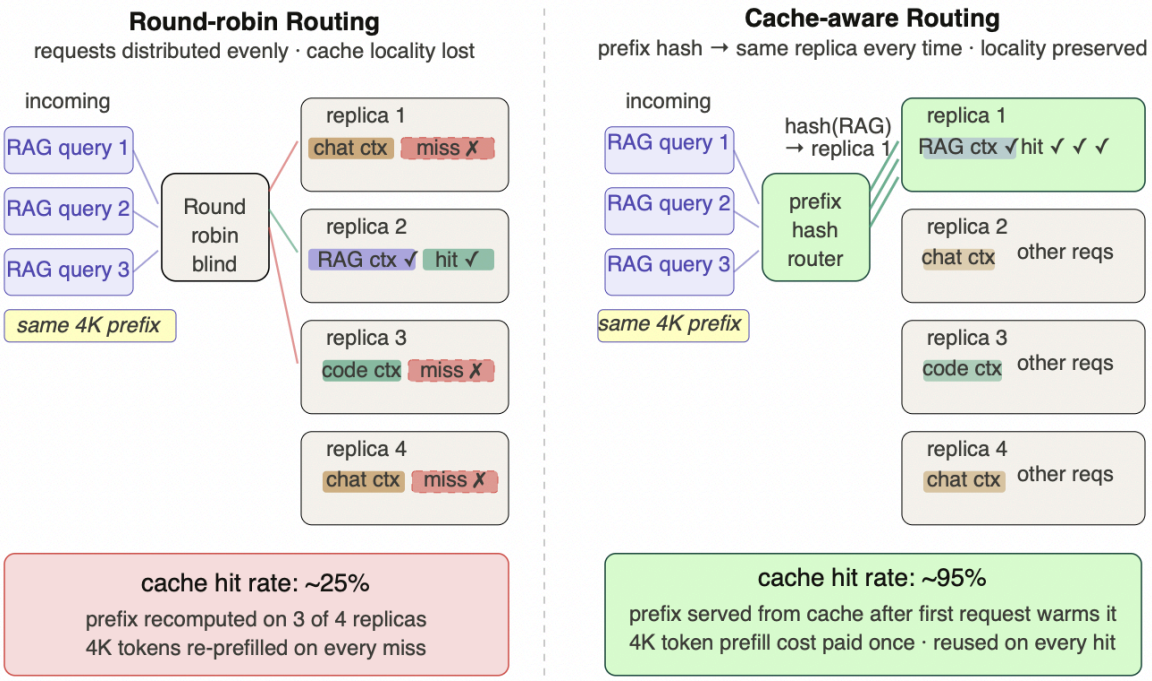

Figure 7.3 - Cache-aware routing: why round-robin kills your cache hit rate
Left: round-robin distributes three requests with identical 4K-token RAG prefixes across four replicas. Each replica holds a different cached prefix - only replica 2 happens to hold the RAG context. Two of the three requests miss, recomputing 4K tokens of prefill from scratch. Hit rate collapses to ~25% regardless of how well the cache is configured. Right: a prefix-hash router computes a deterministic hash of the shared prefix and always routes RAG queries to replica 1. After the first request warms the cache, every subsequent request hits. The 4K prefill cost is paid once and amortized across all requests. The key insight: prefix caching and cache-aware routing are not independent optimizations - one without the other delivers most of neither. This is the most commonly overlooked gap between single-instance benchmarks and multi-replica production deployments.

A note on the routing implementation.
The simplest way to route by prefix is: hash the prefix, divide by the number of replicas, send to that replica - `replica = hash(prefix) % N`. This works until you add or remove a replica. When N changes from 4 to 5, a prefix with hash value 17 that previously went to replica 1 `(17 % 4 = 1)` now goes to replica 2 `(17 % 5 = 2)`. This is not just one prefix - mathematically, only about 1 in N prefixes will accidentally land on the same replica after the change. The other `(N-1)/N` of your prefixes remap simultaneously, every affected replica starts cold, and you get a fleet-wide cache miss storm during what should have been a routine scaling event.

Consistent hashing avoids this by placing both replicas and prefixes on a fixed ring rather than using replica count as the divisor. Each prefix routes to the nearest replica clockwise on the ring. When you add a new replica, it takes a position on the ring and absorbs only the prefixes between it and its neighbor - roughly 1/N of traffic. Everything else stays exactly where it was. Cache hit rates dip briefly for a small fraction of traffic rather than collapsing across the entire fleet.

Before relying on prefix caching in a multi-replica deployment, verify that your serving framework uses consistent hashing for prefix routing - not simple modulo. The distinction is invisible until your first autoscaling event, at which point it becomes very visible very quickly.

The load balance tension is the core design tradeoff.

Prefix-hash routing concentrates traffic - all requests sharing the same popular prefix route to the same replica. For a RAG deployment where 80% of requests share a common system prompt, prefix-hash routing sends 80% of traffic to one replica. That replica becomes a hotspot: memory pressure rises, queue depth grows, and latency degrades precisely for the most common request type. The solution is a hybrid policy - use prefix-hash routing as the primary signal, but cap any single replica's load share and overflow excess traffic to the next available replica. SGLang's cache-aware load balancer implements this with a configurable load threshold; llm-d's router uses a similar overflow mechanism. The threshold is a tunable tradeoff between cache locality and load balance - set it too high and you get hotspots, too low and you sacrifice hit rates.

The cold start problem during scaling and restarts.

When a new replica comes online or an existing one restarts, its cache is cold. Prefix-hash routing immediately sends all matching traffic to that cold replica, producing a burst of cache misses and prefill compute spikes that can briefly saturate the new replica's GPU. Production deployments handle this in two ways: gradual traffic shifting - ramping the new replica from 0% to full share over several minutes to allow the cache to warm under controlled load - or cache seeding, where a peer replica transfers its most frequently accessed prefix blocks to the new replica before it begins receiving traffic. Neither is complex to implement, but both require explicit operational handling that most teams discover only after their first scaling incident.

Prefix caching and cache-aware routing are not independent optimizations. One without the other delivers most of neither. Single-instance benchmarks that show large prefix caching gains will not replicate in multi-replica production unless routing is configured to preserve locality. This is the most commonly overlooked gap between benchmark results and production outcomes for this optimization.

7.4 KV Cache Eviction: Managing Memory Pressure

Prefix caching and cache-aware routing maximize what you keep in the KV cache. Eviction policy determines what happens when you run out of space to keep it - and the wrong policy for your workload pattern can silently undermine everything the previous two sections built.

KV eviction operates at two distinct levels in production: prefix-level eviction, which governs what survives across requests, and token-level eviction, which governs what survives within a single request's generation. Understanding both is necessary because they address different problems, carry different quality risks, and interact differently with other optimizations in your stack.

Prefix-Level Eviction - What Stays in the Cache Across Requests

When the prefix cache fills, the serving system must decide which cached KV blocks to keep and which to discard. Three policy families cover most production scenarios.

LRU - Least Recently Used is the default in vLLM, SGLang, and most production frameworks. The rule is simple: when space is needed, evict the prefix that was accessed least recently. Think of it like a desk where you keep the documents you touched most recently on top and sweep the oldest ones into a drawer when the desk fills up.

LRU works well for diverse, unpredictable workloads where no single prefix dominates traffic and reuse patterns are irregular. It requires no configuration and degrades gracefully as traffic patterns shift.

Where LRU fails is workloads dominated by a small set of high-frequency prefixes. Consider a RAG deployment where 80% of requests share the same 8K-token system prompt. Under bursty diverse traffic, that system prompt can slide to the bottom of the recency queue and get evicted - forcing recomputation of 8K tokens for the next request that needs it, which is immediately. LRU does not distinguish between "this was accessed 10 seconds ago by 1 request" and "this was accessed 10 seconds ago by 1,000 requests." Recency and frequency are different signals, and LRU only tracks one of them.

Frequency-based eviction fixes this by tracking how often each prefix has been reused and protecting the most frequently accessed ones regardless of when they were last accessed. The rule: when space is needed, evict the prefix that has been reused the fewest times overall, not the one that was accessed least recently.

This is the right policy when a small set of prompts - a shared system prompt, a fixed RAG context template, a few-shot preamble - accounts for the majority of your traffic. Those high-frequency prefixes stay permanently resident regardless of recency fluctuations.

Where frequency-based eviction fails is highly diverse workloads where no prefix repeats more than once or twice. With no meaningful frequency signal to act on, the policy degrades toward random eviction behavior and adds bookkeeping overhead without benefit. If your workload's prefix reuse distribution is flat, LRU is the better default.

Priority-based eviction takes a different approach entirely - instead of evicting based on recency or frequency, it protects in-flight and recently active sessions from eviction regardless of their statistical rank. When space is needed, lower-priority background prefixes are evicted first and active session context is preserved.

This matters most for multi-turn conversational deployments. If a user is mid-conversation and the decode instance evicts their conversation history to make room for a new request, the user's next message must recompute the entire conversation from scratch. That recomputation adds latency that is directly user-visible - the model appears to stall before responding. Priority-based eviction prevents this by treating active session context as protected memory that yields only to other active sessions, never to background or cold prefixes.

The practical guidance: start with LRU as your default. If monitoring shows that your most frequently accessed prefixes are being evicted and recomputed repeatedly - you will see this

as elevated prefill compute on requests that should be cache hits - switch to frequency-based eviction. Add priority-based protection for active sessions only if multi-turn latency degradation is measurable in your P99 metrics.

The Memory Pressure Cascade - What Actually Goes Wrong

Rising eviction rate is not a normal operating condition. It is an early warning signal for a failure mode that presents as a latency problem rather than a memory problem, which makes it genuinely difficult to diagnose without the right metrics.

Here is the cascade sequence. The KV cache fills. Eviction begins. Evicted active requests must either be swapped to CPU DRAM or recomputed from scratch when their next token arrives. If the eviction rate exceeds the recovery rate, requests queue behind their own evicted state. Queue depth rises. TTFT grows. GPU utilization drops because the GPU is waiting for recovery operations rather than doing useful work. From the outside, the system looks like it has a latency problem or a throughput problem - not a memory problem.

The practical threshold: eviction rates above 5-10% of active requests per second indicate your KV cache budget is undersized for current traffic. Above 20%, eviction overhead is materially degrading throughput and latency. Treat this as a capacity signal - either provision more KV cache budget, apply more aggressive KV cache quantization, or reduce concurrent load.

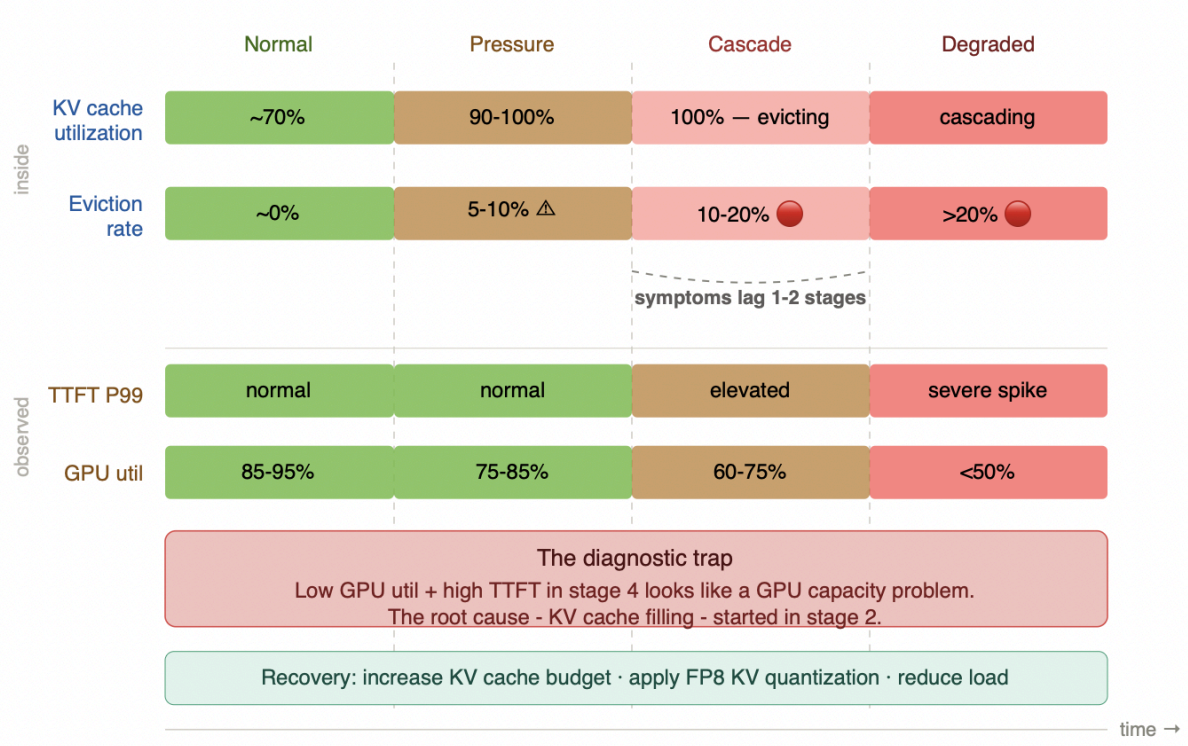

Figure 7.4 - The KV cache memory pressure cascade
The top two rows show what is happening inside the system - KV cache utilization and eviction rate both turn critical by stage 2. The bottom two rows show what is externally observable - TTFT and GPU utilization stay healthy through stage 2 and only degrade in stages 3 and 4. By the time your

latency alerts fire, the root cause has been building for two stages. A system presenting as a GPU capacity problem in stage 4 almost always has a KV cache budget problem that originated in stage 2. Instrument eviction rate from day one - it is the only metric that fires early enough to intervene before the cascade reaches the observable layer.

Swap vs Recompute - The Recovery Decision

When an active request's KV state is evicted, the framework must choose between two recovery paths. This is a decision most teams leave at framework defaults without understanding the tradeoff.

Swapping to CPU DRAM preserves the full KV state by moving it off GPU memory to system RAM. When the request resumes, the KV blocks are transferred back via PCIe. This preserves output quality perfectly - the model sees the same KV state it would have seen without eviction. The cost is transfer latency: at PCIe Gen5 bandwidth of ~64 GB/s per direction, a 1 GB KV cache takes roughly 15ms to swap back. For long-context requests with large KV states, this is usually cheaper than recomputation.

Recomputing from scratch discards the evicted KV state entirely and reruns the prefill on the evicted tokens when the request resumes. This avoids the PCIe transfer but burns GPU compute proportional to the evicted context length. For short contexts - under a few hundred tokens - recomputation is typically faster than swapping. For long contexts - thousands of tokens - swapping is almost always the better choice.

vLLM implements both strategies with a configurable swap threshold. Most deployments should verify this threshold is tuned to their typical context length distribution rather than left at the default. If your workload is predominantly long-context, bias toward swapping. If it is predominantly short-context, recomputation is cheaper.

Eviction in Disaggregated Deployments - A Special Case

In a disaggregated prefill-decode deployment, eviction on decode instances has a consequence that does not exist in colocated systems. If a decode instance evicts a prefix that an incoming request needs, the request cannot recover locally - it must be sent back to a prefill instance for recomputation, adding a full prefill round trip to what should have been a decode-only operation. This breaks the disaggregation latency model for evicted sessions and can make P99 latency significantly worse than a colocated baseline for affected requests.

Two mitigations: first, provision decode instance KV cache budgets conservatively - size for active request concurrency plus a meaningful buffer for frequently reused prefixes, not just the minimum to fit active requests. Second, implement session pinning for active multi-turn conversations so their KV state is marked protected and does not participate in eviction until the session ends.

Token-Level Eviction - Reducing Memory Within a Single Request

Prefix-level eviction manages memory across requests. Token-level eviction manages memory within a single request's generation - selectively dropping individual token positions from the KV cache during generation based on estimated attention importance.

The motivation is the attention skew observation: in transformer models, a small subset of token positions consistently receives the majority of future attention mass. Most tokens, once generated a few steps ago, are rarely attended to again. Storing their full KV state wastes memory that could serve more concurrent requests.

SnapKV estimates which token positions are actively attended to based on recent attention patterns observed during generation and evicts the positions that fall below an attention threshold. It is adaptive - the set of protected positions updates as generation proceeds and the model's attention focus shifts.

H2O (Heavy Hitter Oracle) takes a cumulative approach, tracking total attention scores across all generation steps and permanently protecting the tokens that have received the highest cumulative attention mass - the "heavy hitters." Tokens that have never received significant attention across the entire generation history are candidates for eviction.

Both approaches reduce KV memory footprint meaningfully - reported reductions of 20-50% are realistic for long-context workloads - with quality degradation that is workload-dependent.

For conversational workloads where the model's attention is dominated by recent context and the task does not require retrieving specific information from early in the sequence, token-level eviction produces negligible quality loss. The evicted tokens genuinely are not contributing to the model's generation.

For long-context retrieval tasks - legal document analysis, long-form summarization, needle-in-a-haystack queries, agentic tool-call histories - the evicted tokens may contain exactly the information the model needs to retrieve later in generation. Quality degradation in these cases can be significant and task-specific. Always benchmark on your specific task and workload before enabling token-level eviction in production. The memory saving is real; so is the risk.

One compatibility issue that is not widely documented: token-level eviction modifies the KV cache structure by removing positions that the framework assumes are present. Speculative decoding's verification pass assumes the KV cache is a complete record of all prior token positions - if token-level eviction has removed positions that the draft model speculated against, the verification pass produces incorrect results. Do not combine token-level eviction with speculative decoding without explicitly verifying framework-level compatibility first.

Decision Sequence

The strategies above are not mutually exclusive - they layer on top of each other as your workload complexity and memory pressure grow. The table below maps the common scenarios to the right starting point and tells you exactly when to add the next layer.

Scenario	Policy	Trigger to escalate
Default starting point, diverse unpredictable traffic	LRU prefix eviction	High-frequency prefixes appearing in prefill compute despite being recently cached
Small set of prompts dominates traffic (RAG, shared system prompt)	Frequency-based eviction	LRU eviction rate on your top-5 prefixes is measurable - they keep getting dropped and recomputed
Multi-turn conversational workload	+ Priority-based eviction for active sessions	P99 latency spikes visible on conversation turns that should be cache hits
Eviction rate above 10% of active requests per second	Increase KV cache budget, apply KV cache quantization (Section 7.5), or reduce load	Eviction rate above 20% - system is in degraded state, not a normal operating condition
Disaggregated P/D deployment	Larger decode KV cache budget + session pinning	Any active session being evicted and sent back to prefill pool for recomputation
Long-context workload where KV cache dominates memory budget	Evaluate token-level eviction (SnapKV or H2O)	Only after quality validation on your specific task - do not enable speculatively
Running both token-level eviction and speculative decoding	Verify framework compatibility explicitly	Incompatible by default - token-level eviction removes positions speculative decoding assumes are present

7.5 KV Cache Quantization: A Concurrency Lever

KV cache quantization appeared as a recovery lever in the previous section. It deserves its own treatment because it is not just an emergency response to memory pressure - it is a proactive capacity decision that changes the economics of your deployment before pressure ever builds.

It is also fundamentally different from weight quantization, and the difference matters operationally. Weight quantization reduces model size and compute cost permanently - it changes the model and affects every request's output quality in a fixed, measurable way. KV cache quantization reduces runtime memory consumption per active request, which directly converts saved memory into additional serving capacity. It does not change the model. It

does not affect requests that fit entirely within the quantized cache without overflow. And unlike weight quantization, its quality impact scales with context length - short-context evaluations will not surface the degradation that long-context tasks may experience.

The memory math is the starting point. At FP16, Llama-3 70B generates approximately 0.26 MB of KV cache per token per request. At 50 concurrent requests with 4K-token contexts that is 52 GB - more than half an H100's 80 GB, leaving limited headroom for model weights and activations. FP8 KV cache halves this to 26 GB. Same hardware, same model, same concurrency target - but now you have 26 GB of recovered headroom that can either serve more concurrent requests or support longer contexts. At NVFP4 on Blackwell hardware the savings compound further to 13 GB for the same workload.

The three precision tiers and their tradeoffs.

FP8 KV cache is the production default on H100 and Blackwell and is supported in vLLM and SGLang. Quality impact is minimal for most workloads - empirically, short-to-medium context tasks show negligible degradation. The risk surface is long-context precision-sensitive tasks: legal document analysis, multi-hop reasoning over long context windows, and tasks where the model must retrieve specific information from early in a long sequence. In these cases, the reduced precision in stored key-value vectors can cause the model to attend to slightly different positions, producing subtly different outputs. Always run your specific long-context evaluation suite before declaring FP8 KV cache safe for production on precision-sensitive workloads.

NVFP4 KV cache, available on Blackwell hardware, halves memory again relative to FP8. The quality tradeoff is more pronounced and workload-dependent. On tasks dominated by recent context - conversational, short-output workloads - the degradation remains manageable. On tasks requiring precise retrieval from long histories, quality degradation becomes measurable. Treat NVFP4 KV cache as an aggressive capacity lever for latency-tolerant workloads where maximum concurrency is the primary objective, not a default setting.

KVQuant and similar research approaches apply non-uniform quantization tailored to the statistical distribution of KV tensors rather than fixed-precision formats. Because KV distributions are not uniform - certain attention heads and certain token positions carry disproportionately high information content - adaptive quantization can achieve lower effective bit-widths with less quality loss than naive FP8. These approaches are not yet standard in production frameworks but are worth tracking as they mature.

The composition property is significant. Because KV cache is fully independent of model weights, these optimizations stack cleanly. A deployment running FP8 weights can additionally apply FP8 KV cache quantization, compounding memory savings without any interaction between the two decisions. The total memory reduction from combining weight and KV cache quantization is additive - the two levers do not interfere with each other and do not require coordinated quality validation.

How to sequence the decision. Start with FP8 KV cache as the default on H100 and Blackwell - enable it, validate quality on your specific task distribution including your

longest-context evaluation cases, and keep it on unless you find a specific failure mode. If you need more capacity beyond what FP8 provides, evaluate NVFP4 on Blackwell with a careful quality benchmark. If you are hitting the limits of what fixed-precision quantization can provide without unacceptable quality loss, KVQuant-style adaptive approaches are the frontier to watch.

KV cache quantization addresses the per-request memory footprint. The next lever - offloading - addresses a different constraint: what happens to requests whose KV state cannot fit in GPU memory at all, even after quantization.

7.6 KV Cache Offloading: Trading Latency for Capacity

KV cache quantization reduces per-token memory cost. At long enough context lengths, even quantized KV cache exceeds what GPU HBM can hold for the number of concurrent requests you need to serve. A 100K-token context on Llama-3 70B at FP16 generates roughly 8 GB of KV cache for a single request - at FP8 that is still 4 GB. On an 80 GB H100 with 70 GB consumed by model weights and framework overhead, you can serve exactly one such request before the GPU is full. At long context lengths, KV cache stops behaving like an ephemeral compute state and starts behaving like a dataset - large, structured, and only partially active at any moment.

This is architecturally distinct from the reactive swap discussed in Section 7.4. Eviction-driven swap is a pressure relief valve - it kicks in when the cache fills unexpectedly under load. KV cache offloading is a deliberate tiering decision made at deployment time for workloads where long-context capacity is a known, structural requirement.

KV cache offloading solves the capacity problem by treating GPU memory as the hot tier in a storage hierarchy. Active KV blocks stay in HBM. Blocks that are not immediately needed move to CPU DRAM or NVMe and are fetched back when attention requires them - the same way a database tiers hot and cold data across DRAM and SSD. The GPU's HBM budget is freed to serve more concurrent contexts rather than holding the full KV state of every active request simultaneously.

The cost is reload latency, and the math determines whether it is worth paying.

PCIe Gen5 delivers roughly 64 GB/s effective transfer bandwidth between GPU and CPU DRAM. That 8 GB KV cache takes approximately 125ms to reload from CPU DRAM. NVMe adds a second tier with higher latency - current enterprise NVMe sequential read bandwidth peaks at roughly 7 GB/s, meaning the same 8 GB KV cache takes approximately 1.1 seconds to reload from NVMe. That second tier is only viable for offline batch workloads with no interactive latency SLO.

Whether the CPU DRAM tier is acceptable depends entirely on your workload. For interactive chat, 125ms per cache miss adds visible jitter to token generation - the user sees a stall mid-response every time a block is fetched back. Offloading is the wrong tool for this use case. For batch summarization of long documents, 125ms is acceptable if it lets you serve four times more concurrent contexts on the same hardware - the throughput gain justifies the latency cost, and no user is waiting for individual tokens.

Mooncake - ByteDance's KV cache-centric disaggregated serving architecture - validates this pattern at scale, treating KV storage as a first-class distributed resource separate from compute and demonstrating meaningful throughput improvements in memory-bound regimes by putting otherwise idle CPU DRAM to work systematically rather than reactively.

The decision rule is straightforward. Reach for offloading when the alternative is rejecting requests entirely or OOM-evicting active sessions, and when your latency SLO can absorb the reload cost. For most interactive deployments, KV cache quantization or additional GPU memory is the right first move - both reduce memory pressure without adding reload latency. Offloading is the right answer for long-context batch workloads. It is not the default answer for general serving.

7.7 What to Measure in Production

The optimizations in this section - PagedAttention, prefix caching, cache-aware routing, eviction policy, quantization, offloading - are only as effective as your ability to measure whether they are working. KV cache failures are particularly insidious because they present as latency or throughput problems rather than memory problems. Without the right metrics, you will chase the wrong root cause. The goal is not just monitoring but diagnosis: each metric maps directly to a failure mode and a concrete action.

Metric	What it Measures	Signal	Action
Cache hit rate	Fraction of prefill tokens served from cache vs recomputed	Low hit rate despite known prefix reuse indicates lost locality - missing prefix caching, insufficient cache size, or round-robin routing destroying locality	Enable prefix caching, increase cache budget, or implement cache-aware routing
Prefix cache reuse ratio	Fraction of total prefill compute avoided through caching	High hit rate but low reuse ratio means your cache is hitting on short prefixes that save little compute - the optimization is working but not on the right prefixes	Identify your longest shared prefixes and verify they are being cached and routed correctly
KV cache utilization	Fraction of allocated KV memory actively used	Low utilization → over-provisioning or fragmentation Near 100% → memory-bound, risk of OOM or aggressive eviction	Tune allocation, enable PagedAttention if not already active, or apply KV quantization

Eviction rate	Frequency of cache entries dropped under memory pressure	Rising eviction under steady load means insufficient capacity or wrong eviction policy. Above 10% is a warning. Above 20% is a degraded state	Increase cache budget, switch eviction policy, or reduce footprint via quantization
KV residency time	How long entries stay in cache before eviction	Short residency with high reuse potential indicates cache churn - entries are being computed, cached briefly, evicted, and recomputed	Increase cache size or protect high-frequency entries with frequency-based eviction
KV transfer latency	Time to move KV blocks between tiers - GPU to CPU DRAM in offloading, prefill to decode node in disaggregation	Rising transfer latency in a disaggregated or offloaded deployment signals interconnect saturation, not GPU compute saturation	Check PCIe or network utilization - the bottleneck may be the data path, not the GPU

8 - The Latency-Throughput Curve: Finding Your Operating Point

Every optimization covered in this guide - quantization, batching strategy, parallelism, KV cache tuning - ultimately manifests as a shift in one empirical artifact: the latency-throughput curve for your specific model, hardware, and workload. This is where everything comes together. The curve is not theoretical - it is the empirical basis for every fleet sizing decision, SLO definition, and capacity planning conversation you will have in production.

8.1 The Shape of the Curve and Why the Knee Matters

As concurrency increases, throughput and latency evolve differently. Throughput rises rapidly at first as the system utilizes compute and memory bandwidth more efficiently. Latency remains relatively stable during this phase - requests complete quickly because the system has headroom. As concurrency approaches saturation, something changes: latency begins rising sharply while throughput gains flatten. The system is now spending a growing fraction of its time managing queuing, batching overhead, and resource contention rather than doing useful work.

This shape is not LLM-specific - it is a direct consequence of queueing theory. Any system with finite resources exhibits it. What makes LLM inference distinctive is the steepness of the latency climb past the inflection point, driven by the interaction between prefill compute saturation, decode memory bandwidth limits, and KV cache pressure all converging simultaneously.

The inflection point - where latency begins rising faster than throughput improves - is the knee of the curve. In practice the knee is visible in your monitoring dashboard as the concurrency level where P99 TTFT begins diverging from P50 TTFT. When P50 and P99 track closely together, the system is in the safe operating zone. When P99 starts climbing while P50 stays relatively flat, you are at or past the knee - tail requests are queueing while median requests still complete normally. The knee is not a precise point but a region, and its location shifts with workload characteristics, batch size, and the optimizations you have applied.

Beyond the knee, in the saturation zone, every additional concurrent request buys marginal throughput at a steep latency cost. The system has not broken - it is still serving requests - but the economics have inverted. You are paying in user experience for throughput gains that diminish with every additional request added.

Operating beyond the knee is not just inefficient - it is unstable. Small increases in traffic produce disproportionate latency spikes in the saturation regime. A 10% traffic increase that would be invisible in the safe zone can push P99 TTFT from 500ms to 2 seconds past the

knee. This is why headroom matters: you are not sizing for average traffic, you are sizing for the worst-case burst that your SLO must survive without crossing into saturation.

The production operating point should sit to the left of the knee, with enough margin to absorb traffic spikes. How much margin depends on your traffic variability - a workload with predictable diurnal patterns needs less headroom than one with unpredictable burst events. Section 1's P95 vs average RPS guidance is directly relevant here: if your P95 RPS is 2.5× your average, your safe operating point must be at least 2.5× below the knee's RPS level.

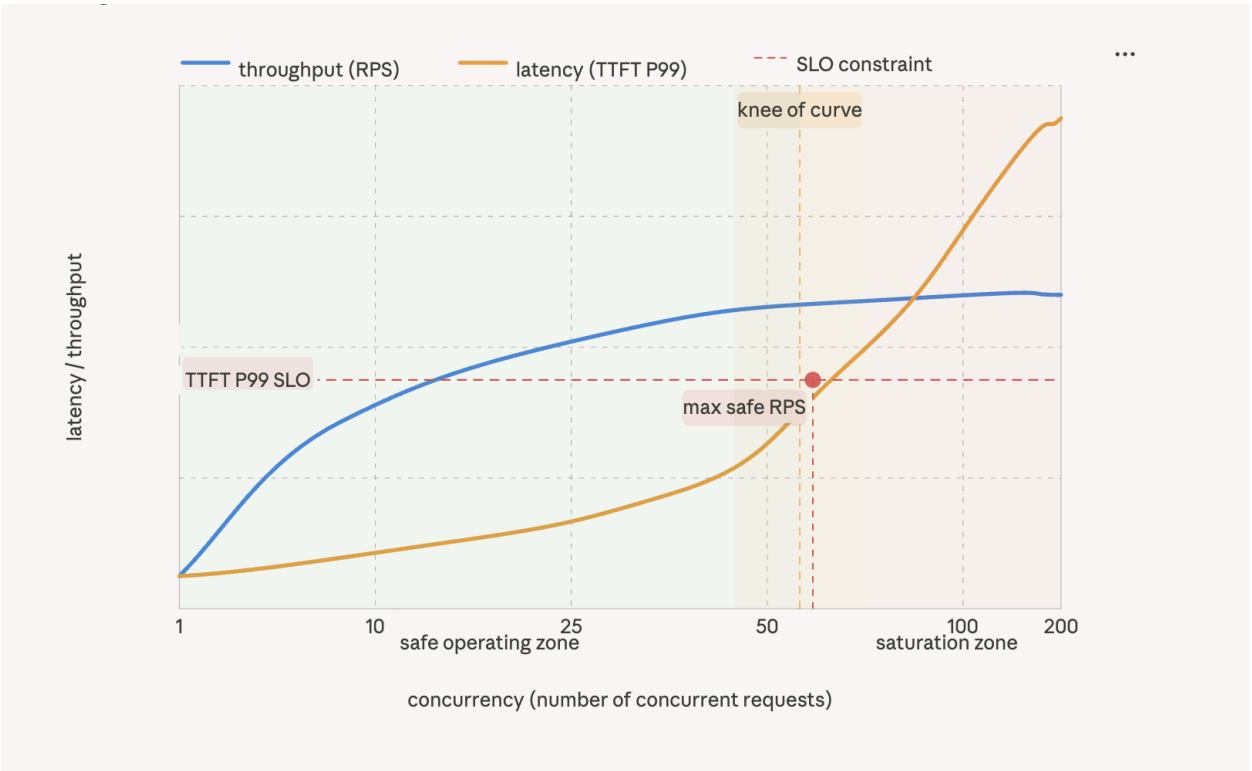

Figure 8.1 - The latency-throughput curve: finding the knee
Throughput (blue) and P99 TTFT latency (amber) respond differently as concurrency increases. In the safe operating zone (left of the knee), throughput grows rapidly while latency stays relatively flat - the system is efficiently utilizing compute and memory bandwidth. At the knee (~60 concurrent requests here), latency begins climbing steeply while throughput gains flatten. Beyond the knee, in the saturation zone, every additional request buys marginal throughput at a steep latency cost as queueing dominates. The red dot marks max safe RPS - the highest concurrency at which P99 TTFT still meets the SLO constraint. This is the production operating point. Sizing the fleet to keep peak traffic to the left of this point - with headroom for spikes - is the central fleet sizing decision.

Understanding the shape of the curve is the first step. Generating it accurately for your specific deployment - model, hardware, workload distribution, and full optimization stack - is the second.

8.2 How to Generate the Curve

The latency-throughput curve cannot be calculated from a formula - it must be measured. Too many variables interact in non-linear ways: KV cache pressure, prefill/decode interference, batching dynamics, and queuing behavior all combine differently at different concurrency levels. The measurement procedure is a concurrency sweep: start at 1 concurrent request, step up to beyond the knee, and record P50 and P99 TTFT, ITL, RPS, and tokens/sec at each step.

Use realistic request distributions, not synthetic uniform load.

This is the single most common mistake in LLM benchmarking. Running a sweep where every request has the same input length and the same output length is like load testing a highway by only sending identical compact cars at uniform spacing. The highway looks fine. Then real traffic arrives - trucks, buses, cars stopping suddenly - and the knee appears much earlier than your benchmark predicted.

Real workloads have heavy-tailed input length distributions. Your P99 input is typically 3-5× your P50 input, and those long-tail requests consume disproportionate KV cache and prefill compute. A sweep measured at uniform 512-token inputs will show a knee at a higher concurrency than your production system actually sustains. Size from your P99 curve, not your P50 curve.

Generate curves for your actual workload archetypes.

There is not one curve for your deployment - there is a family of them, one per input/output length combination. The curve for a RAG workload at 4K input / 200 output looks completely different from a code generation workload at 200 input / 3K output on identical hardware. At minimum generate curves for your P50 and P99 input/output length combinations. If you have multiple workload types on the same fleet, generate one per type.

Also test burst behavior, not just steady-state ramp.

A gradual concurrency ramp shows how the system behaves when load increases slowly. It does not show what happens when 3× normal traffic hits simultaneously - which is exactly what happens during peak events. Run a step-function burst test: send a sudden 2-3× load spike and measure how long TTFT takes to recover. If it takes more than a few seconds to stabilize, your headroom margin is not enough.

Lock down your variables before you start.

A curve is only useful if it is reproducible and comparable. Before running any sweep, explicitly record: model and quantization format, parallelism configuration, hardware, framework version, and input/output length distribution. Benchmark results without these details cannot be used for fleet sizing - they are just numbers without context. This sounds obvious but benchmarks shared without version numbers or length distributions are common and nearly useless.

Always warm up first.

The first few hundred requests in any sweep will show artificially elevated TTFT due to cold KV cache and CUDA initialization. Run at least 1,000 warmup requests and verify TTFT has stabilized before recording measurements.

Tooling.

NVIDIA AIPerf - the successor to GenAI-Perf - supports any OpenAI-compatible endpoint and measures TTFT, ITL, tokens/sec, and RPS directly with configurable concurrency sweeps and length distributions. It is the right starting point for NVIDIA hardware. Watch specifically for where P99 TTFT begins diverging from P50 TTFT - that divergence is the empirical knee, more reliable than looking at throughput alone.

For non-NVIDIA stacks, Locust and k6 work but require custom instrumentation. Standard HTTP response time is not sufficient - you need token-level timestamps for TTFT and ITL specifically, which requires streaming response handling in your load test client.

8.3 From Benchmark to Fleet Size

With the curve in hand, fleet sizing becomes a mechanical process - not an estimate, not a rule of thumb, but a direct calculation from measured data.

The curve gives you two numbers at every concurrency level: latency and throughput. The SLO converts those two numbers into one decision.

Step 1 - Find your SLO-constrained concurrency. On your latency curve, find where TTFT P99 crosses your SLO threshold - say 2 seconds. That concurrency level is **`C_max`**: the highest load your instance can sustain without violating the latency budget. Everything to the right is the region where you are technically serving requests but breaking your SLO at the tail.

Step 2 - Read per-instance RPS at `C_max`. Look at the throughput curve at the same concurrency. That value - not the peak throughput the curve ever reaches - is your usable RPS per instance. The distinction matters: the system may be capable of higher throughput at higher concurrency, but that throughput comes with latency you have agreed not to deliver.

Step 3 - Size the fleet.

```
instances = ceil(peak_RPS / usable_RPS_per_instance)
total_GPUs = instances × TP_degree × safety_factor
```

Add a safety buffer of 20-50% on top of the calculated capacity. This is not conservative padding - it accounts for real-world conditions. Peak traffic estimates are usually based on P95 and are routinely exceeded. Rolling restarts temporarily take instances offline. Operating exactly at `C_max` leaves no margin - any variability such as longer prompts, cache misses, or scheduling delays can push the system into SLO violations. Use 20% for stable, predictable workloads with well-understood traffic patterns. Use 50% for new deployments

without historical traffic data, workloads with high input length variance, or any deployment where an SLO violation is operationally severe.

The failure mode to avoid. The most common capacity planning mistake is sizing for average throughput - what the system delivers at P50 - rather than SLO-constrained throughput at `C_max`. The result is a system that looks healthy on dashboards, passes load tests at average traffic, and then violates P99 SLOs during peak hours - exactly when failures are most visible and most costly. The curve makes this failure mode visible before it reaches production. Little's Law gives you the sanity check to confirm your benchmark is actually telling you the truth about production conditions.

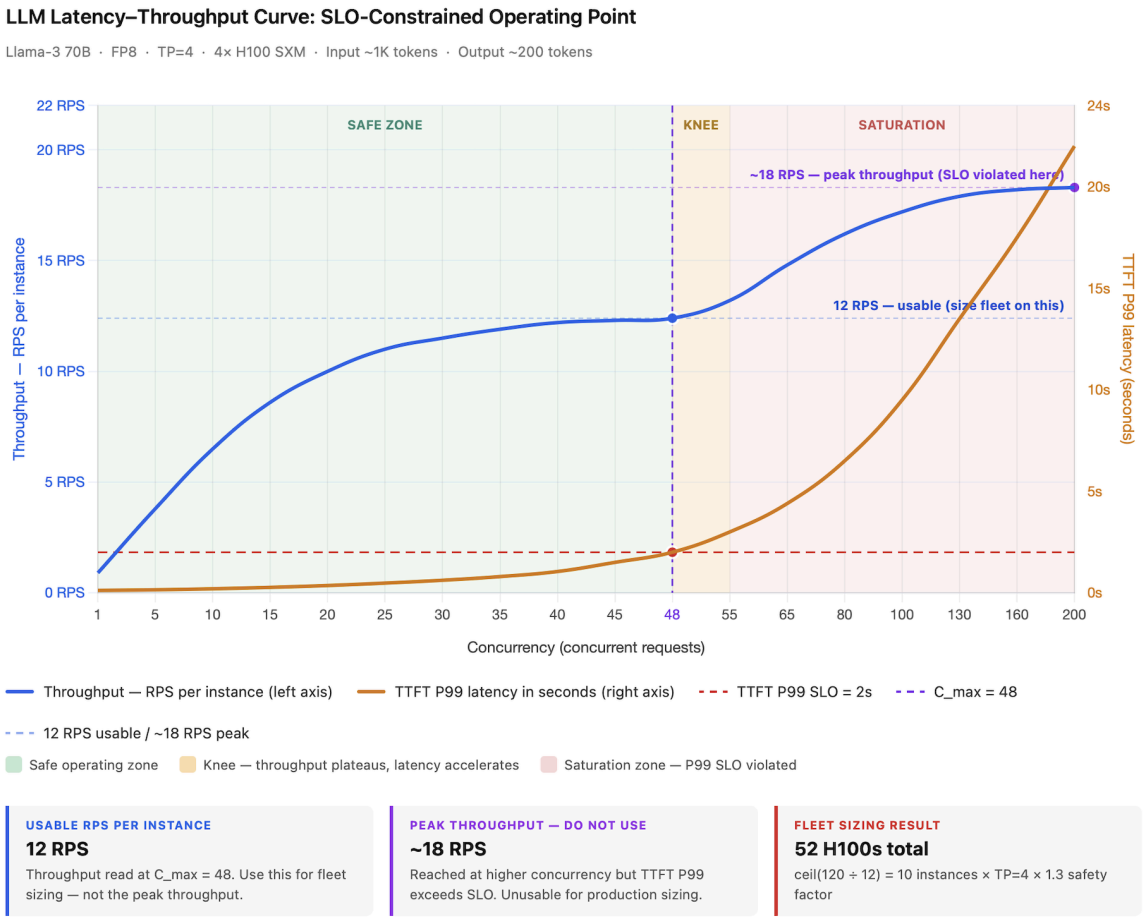

Figure 8.2 - SLO-constrained operating point: from benchmark to fleet size
Llama-3 70B · FP8 · TP=4 · 4× H100 SXM · ~1K input / ~200 output tokens. The TTFT P99 SLO of 2s (red dashed line) intersects the latency curve at C_max = 48 concurrent requests (purple dashed line) - this is the hard ceiling. Everything to the right is the saturation zone where the system is technically serving requests but breaking the latency contract at the tail. Reading throughput at C_max gives 12 RPS usable per instance - not the ~18 RPS peak the curve eventually reaches, which is unreachable without SLO violation. Fleet sizing follows directly: ceil(120 peak RPS ÷ 12 usable RPS) = 10 instances × TP=4 × 1.3 safety factor = 52 H100s total. The gap between the 12 RPS measured and the 24 RPS theoretical ceiling (C_max ÷ SLO = 48 ÷ 2) is expected - the theoretical formula assumes every request takes exactly 2s, while real traffic has variable prompt lengths, cache misses, and

scheduling overhead. A gap under 2× means the benchmark reflects production conditions well. A gap over 2× means the benchmark inputs are wrong and the fleet will be under-provisioned.

Sanity check with Little's Law. Little's Law gives you a theoretical ceiling to validate your benchmark against:

```
maximum_sustainable_RPS = C_max ÷ average latency
```

Plugging in the example: at `C_max = 48` and a 2s SLO, the theoretical ceiling is `48 ÷ 2 = 24 RPS`. The benchmark measured 12 RPS - roughly half.

That gap is not a problem. It is expected. The theoretical formula assumes every request takes exactly 2s. In reality, some requests are longer, some shorter, and the system spends real time on scheduling, KV cache management, and prefill interference. The 24 RPS ceiling is the best case; 12 RPS is what you actually get after system overhead.

Use the gap as a calibration signal. If the gap is under 2×, your benchmark reflects production conditions well and you can proceed with sizing. If the gap exceeds 2×, your benchmark inputs do not match production - the most common cause is test prompt lengths or arrival patterns that differ from real traffic. Fix the benchmark before sizing the fleet. A fleet sized from a mismatched benchmark will be under-provisioned, and the under-provisioning will only surface during peak hours when it is most costly to discover.

Why operating beyond the ceiling always fails. Once the arrival rate exceeds `C_max ÷ average latency`, the system cannot process requests fast enough. The only way it re-establishes balance is by letting latency grow - requests wait longer, the equation rebalances at a higher latency. This is not a bug. It is a mathematical property of any queuing system. You cannot run past this ceiling without breaking your SLO. Sizing from `C_max` is how you enforce that ceiling proactively.

8.4 Start With the Question, Not the Metric

Fleet sizing tells you how many instances you need. The metric you use to track and compare performance determines whether you are measuring the right thing - and the wrong metric is one of the most common sources of misleading benchmark comparisons and bad procurement decisions.

The instinct when evaluating system performance is to lead with metrics - tokens/sec, RPS, utilization - and then work backward to conclusions. This gets the direction wrong. The right approach is to start with the business or operational question you are actually trying to answer, then identify which metric answers it precisely. The same system can look excellent on one metric and poor on another - not because the system changed, but because the question changed. Using tokens/sec/GPU to answer a cost efficiency question will give you a number that looks reasonable and leads you to the wrong fleet size.

Think of it like measuring a delivery service. If your question is "how fast are my trucks?" you measure speed. If your question is "how much am I spending per package delivered?" you

measure cost per delivery. If your question is "are my customers receiving packages on time?" you measure on-time delivery rate. All three use the same trucks and the same routes - but the metric changes completely based on what you need to know. The same principle applies here.

One rule that applies across every row in the table below: never report any metric without also reporting the input/output length distribution it was measured at. A tokens/sec/GPU number at 128-token inputs is not comparable to one at 4K inputs on identical hardware. Vendor benchmark numbers and framework comparison posts routinely omit input/output length distributions - which means the headline tokens/sec figure tells you how fast the system ran on someone else's workload, not yours. Always verify what length distribution a published benchmark used before drawing any conclusions for your deployment.

Business / Operational Question	Why it matters	Metrics that answer it
Is my hardware being used efficiently?	Determines whether optimization headroom exists before buying more GPUs - low efficiency means software changes can still help; high efficiency means hardware is the only remaining lever	**`MFU (Model FLOP Utilization)`** - ratio of achieved to theoretical peak FLOP/s. **`Tokens/sec/GPU`** - strips out instance count and parallelism degree for pure hardware efficiency comparison across frameworks or quantization strategies
Am I serving users within my latency contract?	The only question that directly maps to user experience and SLO compliance - throughput without this is meaningless	**`Goodput`** (SLO-compliant RPS) - counts only requests completed within SLO. A system serving more requests but violating more SLOs has high throughput and low goodput. Always track alongside raw RPS. **`TTFT P99`** and **`ITL P99`** - the latency dimensions of the SLO contract
How many requests can each routing unit handle?	Determines autoscaling thresholds and load balancer configuration - your load balancer routes to instances, not GPUs	**`RPS/instance`** - the unit your infrastructure actually scales and routes on. Tokens/sec/GPU is invisible to your load balancer

What am I getting per dollar of GPU spend?	The cost efficiency question for fleet sizing and procurement - collapses hardware configuration into a single comparable number	**RPS/GPU** - collapses TP degree and instance count into a cost-normalized figure. A TP=2 deployment may beat TP=8 on RPS/instance but lose on RPS/GPU once you account for GPU count. **TTFT per dollar** - adds the latency dimension to cost, essential when comparing GPU generations or cloud providers where price-per-GPU differs
How much of my capacity is doing useful work right now?	Real-time operational question - the leading indicator of approaching the knee before latency metrics start moving. By the time TTFT rises, you are already past the knee	**Queue depth** and **concurrency utilization** - move before latency moves, giving you advance warning. **GPU utilization** - distinguishes between GPU idle (system under-loaded) and GPU busy but inefficient (wrong bottleneck, e.g. memory-bound decode with idle CUDA cores)
Is my system bottlenecked on decode or prefill?	Determines which optimization lever to reach for - decode bottleneck calls for larger batch sizes, higher bandwidth, or speculative decoding; prefill bottleneck calls for more compute or chunked prefill	**TTFT vs ITL ratio** - high TTFT relative to ITL signals prefill is the bottleneck; high ITL relative to TTFT signals decode. **Output tokens/sec vs input tokens/sec ratio** - a system spending most time on decode shows disproportionately high cost per output token relative to input token
How efficiently am I using my memory budget?	Relevant when KV cache dominates - long contexts, high concurrency, or when evaluating quantization strategies for memory impact	**Tokens/sec per GB HBM** - captures the memory efficiency dimension that tokens/sec/GPU misses at long contexts. **KV cache utilization** - what fraction of allocated KV memory is actively used versus fragmented or empty
Is my benchmark representative of production?	Determines whether fleet sizing from the benchmark will be accurate or systematically wrong - a mismatched benchmark produces a fleet that	**Little's Law gap** - compare your measured benchmark RPS against the theoretical ceiling of **C_max ÷ average latency**. Gap under 2× means the benchmark reflects production conditions. Gap over 2× means your test inputs do not match real

	under-provisions exactly at peak load	traffic - fix the benchmark before sizing the fleet
How do I compare two hardware or framework configurations fairly?	Procurement and architecture decisions require apples-to-apples comparisons - most public benchmarks fail this test	`Tokens/sec/GPU` at matched input/output length distributions and matched batch size. Any comparison that does not hold these variables fixed is comparing different workloads, not different hardware. Always report the full configuration alongside the number

The metrics above tell you the current state of your system. What they do not tell you is how a specific optimization - quantization, batching strategy change, parallelism reconfiguration - shifts the curve itself. That is what Section 8.5 covers.

8.5 How Optimizations Shift the Curve

Tracking the right metrics tells you where you are on the curve. This section tells you which lever moves it - and in which direction.

Without the curve, optimization decisions are guesses. With it, every lever has a predictable, measurable effect. Apply an optimization, re-benchmark, and the curve tells you whether it worked and by how much. If the curve did not shift in the expected direction, something else is the binding constraint - and the curve tells you that too.

Optimization	Effect on curve	When to reach for it	What it does not fix
Continuous batching	The single largest curve shift available - moves the knee dramatically rightward by eliminating idle GPU slots between requests. Transforms ~55% GPU utilization to ~95% on the same hardware. Non-negotiable baseline.	Every deployment. If your serving stack does not support it, no other optimization in this table is as effective.	Does not address the prefill/decode interference problem - KV cache pressure and ITL spikes under mixed loads remain.
PagedAttention	Shifts the curve right by eliminating the 20-38% memory waste from contiguous KV allocation. Increases the concurrency the system sustains before	Every deployment - it is the default in all major frameworks. If you are not using it, you are wasting	Does not solve cross-GPU memory imbalance in TP deployments - the most memory-pressured GPU is still the bottleneck.

	hitting the memory ceiling without adding hardware.	your KV cache budget.	
Quantization (FP16 → FP8)	Shifts the entire curve right - more RPS at the same latency, or same RPS at lower latency. Knee moves right too, so you can sustain higher concurrency before saturation.	GPU memory is the binding constraint; you need more concurrent requests per GPU without adding hardware.	Does not help if your bottleneck is compute saturation rather than memory capacity.
Model routing / cascading	Does not shift the large model's curve - reduces traffic volume reaching it. The large model fleet operates at lower effective RPS, moving your operating point left on its curve and improving utilization headroom.	Traffic has measurable complexity skew - many simple requests a smaller model handles adequately. Large model fleet exceeds 8 GPUs.	Does not improve large model latency or throughput. Adds router latency (5-20ms). Misclassification degrades quality on complex requests routed to the small model.
Multi-LoRA serving	No direct effect on curve shape - the base model performance is unchanged. The effect is on fleet economics: one GPU fleet serves N fine-tuned variants instead of N separate fleets, dramatically improving fleet utilization and reducing total GPU count.	Serving multiple fine-tuned variants of the same base model for different tenants or use cases. Without Multi-LoRA you would need a separate fleet per variant.	Does not improve latency or throughput for any individual variant. If simultaneous adapter count exceeds GPU memory budget, adapter paging adds latency that looks like prefill interference but has a different cause.
Speculative decoding	Lowers ITL at low-to-medium concurrency by producing multiple tokens per target model forward pass - extends the flat latency region leftward. No effect on peak RPS. At high concurrency the curve can worsen as verification overhead compounds.	Latency-sensitive workloads at low-to-medium concurrency where decode ITL is the bottleneck and a suitable draft model or n-gram pattern exists.	Does not help at high concurrency or throughput-dominated workloads. Acceptance rate must be measured - a poor draft model match makes it slower than vanilla decode.

KV cache quantization (FP8/INT8 KV)	Independently shifts the curve right by increasing the number of concurrent requests that fit in GPU memory - orthogonal to model weight quantization.	High concurrency workloads where KV cache is the memory ceiling, not model weights.	Does not reduce per-token compute cost.
KV cache offloading	Extends the concurrency range beyond what GPU HBM alone supports by tiering cold KV blocks to CPU DRAM or NVMe. Trades a latency floor increase for offloaded requests in exchange for higher total concurrency.	Long-context batch workloads where KV cache structurally exceeds GPU HBM capacity. Not suitable for interactive workloads.	Adds 125ms+ reload latency per cache miss from CPU DRAM. NVMe tier adds ~1s per miss - only viable for offline batch with no interactive SLO.
KV cache eviction policy	Right policy maintains prefix cache hit rate under memory pressure, preventing TTFT curve degradation as load increases. Wrong policy causes hit rate collapse under load, which appears as a curve shape change rather than a capacity problem.	Any deployment relying on prefix caching at high utilization. LRU is the right default; switch to frequency-based if high-frequency prefixes are repeatedly evicted.	Does not increase total KV cache capacity - it only determines which entries survive when capacity is exhausted.
Prefix caching (high hit rate)	Compresses the effective TTFT curve downward for cache-hit requests - extends the flat latency region significantly. Does not change peak throughput but dramatically improves perceived latency at the same concurrency.	Workloads with shared system prompts, RAG context, or repeated prefixes - multi-turn chat, agentic pipelines.	No benefit for workloads where every request has a unique prefix. Cache hit rate must be measured - assumed hit rates are unreliable.

Cache-aware routing	Restores prefix cache hit rate in multi-replica deployments - without it, round-robin routing collapses hit rate to ~1/N regardless of cache configuration. Enables the prefix caching curve benefit to survive horizontal scaling.	Any multi-replica deployment relying on prefix caching for TTFT improvement. Single-instance deployments do not need it.	Does not help if your workload has no prefix reuse. Creates load imbalance risk if popular prefixes concentrate traffic on one replica.
SLO-aware scheduling	Does not shift the curve itself - changes which requests operate in the flat region versus the saturation region. Protects high-priority requests from tail latency degradation while allowing lower-priority traffic to absorb saturation.	Mixed workloads with different latency classes - interactive chat, batch summarization, and RAG pipelines sharing a fleet.	Does not increase overall capacity. A system at full saturation violates SLOs regardless of scheduling policy.
Chunked prefill	Flattens the ITL curve under mixed workloads - reduces the latency spikes that appear when large prefills enter the batch. Does not increase peak RPS but stabilizes the shape of the curve.	ITL P99 is violating SLO under mixed input lengths even though average ITL is fine.	Does not help TTFT - may slightly increase TTFT for the chunked request itself since prefill is spread across iterations.
P/D disaggregation	Replaces one curve with two independent curves - one for prefill, one for decode - each with its own knee and SLO operating point. Allows each phase to be sized and optimized independently.	Large fleet with strict, independent TTFT and ITL SLOs that cannot both be met on a colocated system.	Adds operational complexity and KV transfer overhead. Net negative for short prompts where transfer cost exceeds interference savings.

Speculative decoding	Lowers ITL at low-to-medium concurrency by producing multiple tokens per forward pass - extends the flat latency region leftward. No effect on peak RPS. At high concurrency the curve can worsen as verification overhead compounds.	Latency-sensitive workloads at low-to-medium concurrency where decode ITL is the bottleneck and a suitable draft model or n-gram pattern exists.	Does not help at high concurrency or throughput-dominated workloads. Acceptance rate must be measured - poor draft model match makes it slower than vanilla decode.
Data parallelism (horizontal scaling)	Duplicates the curve horizontally - multiplies peak RPS linearly with replica count at zero latency overhead. The cleanest scaling lever: doubles replicas, doubles capacity, no tradeoffs.	Per-instance performance is already optimized and you need more aggregate throughput. The last lever to reach for, not the first.	Does not improve per-instance latency or throughput. Each replica must hold a complete model copy - does not reduce per-GPU memory requirements.
Larger TP degree	Lowers per-request latency at low concurrency by parallelizing computation across more GPUs. At high concurrency, all-reduce overhead grows and the knee moves left per replica - you reach saturation earlier with fewer replicas available to absorb traffic spikes.	Latency-sensitive workloads at low to medium concurrency where TTFT or ITL is the binding constraint.	Reduces RPS/GPU - more GPUs per instance means fewer replicas for the same hardware budget. Verify RPS/GPU does not regress.
Expert parallelism (MoE models)	For MoE architectures, EP enables serving models that would not otherwise fit on available hardware, shifting the curve from non-existent to viable. At scale, Wide-EP improves throughput by distributing expert routing across a larger GPU pool.	Any MoE model deployment at scale. For DeepSeek-V3 class models, EP is not optional - it is the mechanism that makes the economics viable.	All-to-all communication overhead adds latency that does not compress well. EP is primarily a throughput optimization, not a latency optimization. Hot expert imbalance can cap throughput well below theoretical maximum.

Higher input/output length	Shifts the entire curve left - lower RPS per GPU, earlier saturation, and steeper latency rise. Not an optimization but a workload reality that must be accounted for in sizing.	N/A - this is a workload characteristic, not a lever. Re-benchmark when your input/output distribution changes materially.	Nothing fixes this except more hardware or a smaller/quantized model.

How to use this table in practice. Start from your specific symptom on the curve, not from a list of optimizations to apply:

- **TTFT too high at low concurrency** → check prefix cache hit rate first, then TP degree
- **ITL spikes under mixed load** → chunked prefill is the first intervention
- **Throughput plateau too low** → quantization (model weights + KV cache) before adding hardware
- **Cannot meet TTFT and ITL SLOs simultaneously** → P/D disaggregation
- **RPS/GPU degrading as you scale TP** → you have passed the all-reduce crossover point; reduce TP and add replicas instead
- **ITL too high at low concurrency with low GPU compute utilization** → speculative decoding before reaching for more hardware
- **TTFT spikes on specific tenants but not others, with no prefill pattern** → check adapter cache eviction rate; an evicted LoRA adapter being paged back from CPU is the likely cause
- **P99 latency healthy at low load but spikes at peak** → you are operating too close to the knee; increase fleet size or apply quantization to move the knee right before the next peak event

The optimizations and levers above address individual components of the system. Section 9 puts them together into a complete sizing algorithm - a sequenced decision process that takes you from workload characterization to a production-ready fleet configuration.

9 - Putting It All Together: Sizing Algorithm and Production Monitoring

Section 8 showed how to read the latency-throughput curve and size from it. Every section before that covered an individual lever. This section assembles the full decision sequence - the order in which those levers must be pulled, why the order matters, and how to verify in production that the result is correct.

9.1 The Sizing Algorithm: Decision Sequence and Dependencies

The order of decisions matters because each step constrains the next. Getting the sequence wrong means re-doing work. The most common mistake: selecting parallelism degree before quantizing. Quantization changes the memory floor - which determines whether the model fits on N GPUs or requires more. If you select TP degree against an FP16 memory floor and then quantize to FP8, your TP choice was made against a constraint that no longer exists. You may have chosen TP=4 to fit the model when TP=2 would have been sufficient after quantization - with twice the replicas, half the all-reduce overhead, and better aggregate throughput.

Step	Decision	Key output	Why this order
1	**Characterize workload** - measure or estimate P50/P95/P99 input and output lengths, peak RPS vs average RPS, TTFT and ITL SLO targets, online streaming vs offline batch split, and traffic complexity distribution. Reference Section 1 in full - this step is underestimated more than any other.	Workload profile: length distributions, peak load, SLO targets, workload archetype	Nothing else can be calculated without this. Every downstream decision is parameterized by these inputs. Errors here propagate through every subsequent step.

2	**Hardware selection** - based on workload archetype from Step 1, select GPU optimizing for TFLOPS (prefill-heavy: H100/B200) or HBM bandwidth (decode-heavy: MI300X/H200). Verify interconnect: NVLink required for TP > 2; PCIe limits viable TP degree.	Target GPU model, interconnect topology	Must precede memory sizing - memory floor and parallelism options both depend on GPU HBM capacity and bandwidth. Wrong hardware selection cannot be fixed by software optimization.
3	**Routing decision** - does traffic complexity skew justify model routing or cascading? Evaluate if the large model fleet would exceed 8 GPUs and traffic has measurable simple-request skew. If yes, define tier boundaries and run Steps 4-10 independently per tier.	Single-fleet or multi-tier architecture	Must precede memory sizing - routing determines how many fleets you are sizing and what traffic volume each fleet sees.
4	**Calculate memory floor** - model weights at target precision + KV cache at P95 concurrency and P95 input length + activations (~20% of weight memory) + framework overhead (5–10% of total VRAM) + adapter memory if serving LoRA variants (max_simultaneous_adapters × adapter_size_at_target_rank).	Minimum GPU count before optimization	Establishes the hard memory constraint. Cannot be estimated - must be calculated from actual workload parameters.
5	**Apply quantization and sparsity** - select precision (FP8 recommended default on H100/Blackwell, INT4 via AWQ for maximum compression on tolerant workloads). Apply KV cache quantization independently. Evaluate sparsity (2:4 structured) only if compute throughput on prefill-heavy workloads remains the binding constraint after weight quantization.	Reduced memory footprint, revised minimum GPU count, revised compute profile	Must precede parallelism selection - quantization often moves from 2 nodes to 1 or 8 GPUs to 4, completely changing the parallelism design space. Applying after parallelism selection wastes the sizing work.

6	**Select parallelism strategy** - TP degree to fit model within a node (lowest degree that fits after Step 5), PP only if model cannot fit within single node's NVLink domain, DP replica count deferred to Step 9. Verify all-reduce overhead is acceptable at chosen TP degree on available interconnect.	TP degree, PP degree if needed, interconnect validation	Depends on memory floor after quantization (Step 5) and latency SLO (Step 1). TP degree determines replica count and therefore aggregate throughput capacity.
7	**Benchmark single instance** - run concurrency sweep with realistic P50/P95 input/output length distribution. Record P50/P99 TTFT and ITL alongside RPS and tokens/sec. Identify C_max (SLO-constrained concurrency) and usable RPS per instance. Validate with Little's Law - gap over 2× means benchmark inputs do not match production.	Latency-throughput curve, usable RPS per instance at SLO	Must use actual configuration from Steps 5+6. Benchmarking unquantized or differently parallelized configuration produces a curve that does not reflect production behavior.
8	**Configure KV cache strategy** - in order: verify PagedAttention is active (default in all major frameworks), apply KV cache quantization (FP8 default, NVFP4 on Blackwell for maximum concurrency), enable prefix caching if workload has shared prefixes, configure cache-aware routing for multi-replica deployments, set eviction policy (LRU default; frequency-based if high-frequency prefixes are repeatedly evicted), evaluate KV cache offloading only for long-context batch workloads where KV cache structurally exceeds GPU HBM after quantization.	KV cache hit rate, effective concurrency headroom, eviction rate	After baseline benchmark - these optimizations shift the curve and the Step 7 benchmark establishes the baseline to measure improvement against.

9	**Configure batching and decode acceleration** - in order: verify continuous batching is active, configure SLO-aware scheduler for mixed workloads, evaluate chunked prefill if ITL P99 spikes under mixed input lengths, evaluate speculative decoding for latency-sensitive low-concurrency workloads with measurable acceptance rate, evaluate P/D disaggregation for large fleets where TTFT and ITL SLOs cannot simultaneously be met on colocated system.	Final serving configuration	After Step 8 - KV cache strategy affects which batching configurations are viable. Disaggregation decision depends on whether KV cache optimization alone resolves the TTFT/ITL tension.
10	**Re-benchmark with full optimization stack** - repeat Step 7 with all optimizations from Steps 8+9 active. Each optimization shifts the curve - the Step 7 baseline is no longer valid. Fleet size must be calculated from the final curve.	Final latency-throughput curve, final usable RPS per instance	The Step 7 benchmark is a baseline, not the final number. Skipping this step means sizing from a curve that does not reflect the actual production configuration.
11	**Calculate fleet size and DP degree** - `ceil(peak_RPS / usable_RPS_per_instance) × safety factor (20% for predictable workloads, 50% for new deployments or high variance)`. Total GPUs = instances × TP_degree.	Total instance count, total GPU count, DP degree	Depends on Step 10 (final usable RPS) and Step 1 (peak RPS target and traffic variance).
12	**Calculate TCO** - GPU count × GPU hourly rate ÷ utilization-adjusted token output rate. Average utilization - not peak - determines real cost per token. A fleet sized for P95 peak running at 40% average utilization costs 2× per token versus one running at 80%.	Cost per token, utilization-adjusted annual cost	Depends on Step 11 (GPU count) and measured or estimated average utilization.

13	**Deploy and validate** - within the first week of production traffic, compare actual TTFT P99, ITL P99, GPU utilization, KV cache hit rate, and eviction rate against Step 10 benchmark predictions. If production TTFT P99 runs more than 30% above benchmark, or GPU utilization runs more than 20 points below benchmark at equivalent RPS - return to Step 1 and re-characterize the workload. Production traffic almost always differs from benchmark assumptions in input length distribution or arrival patterns. Section 9.3 covers the full ongoing monitoring framework.	Validation that sizing matches real traffic. Trigger for re-sizing if divergence is significant.	The algorithm is complete only when production data confirms the benchmark was representative. Sizing is not a one-time event - it closes a loop back to Step 1.

Average utilization - not peak - is the hidden variable in TCO. A fleet sized correctly for peak traffic can still have 2-3× higher cost per token than expected due to off-peak idle time. Section 9.2 covers this in full.

9.2 The Utilization Trap: The Hidden Variable in TCO

GPU count from the sizing algorithm tells you the hardware you need at peak load. TCO depends on a different question: *what fraction of that hardware is doing useful work on average?*

Think of it like an airline. A plane flying at 40% seat occupancy pays full fuel and crew costs regardless - empty seats do not get a discount. GPU fleets have exactly the same dynamic. You pay for the GPU whether it is serving a request or sitting idle. The GPU does not know the difference. Your invoice does not either.

Most production LLM deployments run at 30-60% average GPU utilization due to traffic burstiness - high during business hours, near-zero overnight. A fleet sized correctly for P95 peak but running at 40% average utilization is paying 2.5× the efficient cost per token compared to a fleet running at full utilization on the same hardware.

```
Cost per token = (GPU-hour cost × GPU count) ÷ (tokens
generated × utilization rate)
```

For most deployments running below 60% average utilization - which describes the majority of production LLM fleets - utilization rate has more leverage on cost per token than any infrastructure optimization. A jump from 40% to 70% utilization cuts cost per token nearly in half without touching the model, the serving stack, or the hardware. Quantization saves memory. Batching improves throughput. Utilization determines what fraction of your bill is producing anything at all.

Levers to improve utilization, in rough order of operational complexity:

Model colocation shares GPU capacity across multiple models or tenants on the same fleet. NVIDIA Multi-Instance GPU (MIG) partitions a single H100 into up to seven independent GPU instances each with dedicated HBM and compute - allowing multiple smaller models or tenants to share one physical GPU without memory interference or noisy-neighbor effects. Time-sharing without MIG is also viable for latency-tolerant workloads where tenants do not need guaranteed isolation. Either approach converts a fleet serving one model at 40% utilization into one serving multiple workloads at 70–80%.

Offline batch fill routes non-latency-sensitive jobs - nightly summarization, embedding generation, dataset annotation, fine-tuning data preparation - to fill idle capacity during off-peak hours. The fleet is already warm and already paid for. Offline batch work at 2am costs almost nothing marginal beyond the incremental electricity. This is the easiest utilization lever for any team that has batch workloads sitting in a queue.

Aggressive autoscaling scales down faster than feels comfortable. The cost of over-scaling - paying for idle GPUs overnight - almost always dwarfs the occasional cold-start latency penalty. Most teams scale down too conservatively because cold starts feel expensive and risky. But a fleet running at 30% utilization overnight is paying 3.3× per token for the comfort of instant scale-up readiness. The math rarely justifies it.

Request-level token reduction trims system prompts, caps max output tokens, and compresses RAG context. Fewer tokens per request means more requests per GPU-hour - the same fleet serves more traffic without adding hardware. This is an application-level lever, not an infrastructure lever, and it is often the highest-ROI intervention available to teams that have already optimized their serving stack. A 30% reduction in average prompt length translates directly to a 30% improvement in effective throughput at constant GPU count.

Model routing directs simple requests to a small model on commodity hardware, reserving large model GPU capacity for requests that genuinely need it. This is the only lever on this list that reduces large model traffic volume rather than filling idle capacity - it improves utilization economics by shrinking the fleet that needs to be kept warm, not by filling gaps in an oversized one. A large model fleet serving 40% simple requests that a 7B model could handle is paying frontier model rates for commodity work.

A note on LLM autoscaling mechanics - why reactive scaling fails and what to do instead.

LLM serving instances have GPU cold start times of 2–5 minutes - the time required to load model weights from NVMe storage into GPU HBM. A 70B FP8 model loading at typical NVMe bandwidth takes 30-60 seconds for the transfer alone, before the serving framework initializes. This is orders of magnitude longer than web service cold starts, which typically complete in seconds.

Reactive autoscaling - scale up when GPU utilization crosses a threshold - responds too slowly for bursty LLM traffic. The sequence plays out the same way every time: traffic spikes, utilization threshold is crossed, autoscaler triggers new instance provisioning, 2-5 minutes pass, new instances become ready, traffic spike has already peaked and is subsiding, SLO violations have already occurred. The autoscaler fixed a problem that is already over.

The production pattern is predictive scaling. Use historical traffic patterns - time-of-day curves, day-of-week seasonality, known event spikes - to pre-warm instances before anticipated peaks rather than reacting to them. Kubernetes HPA on GPU utilization alone is insufficient for LLM fleets. The right architecture pairs HPA with two additional layers: scheduled scaling rules that pre-warm instances ahead of known traffic patterns, and a minimum warm instance count that guarantees at least one instance is always available to absorb sudden unscheduled spikes without waiting for cold start.

The minimum warm instance count is not waste - it is the premium you pay for responsive scaling. At most traffic levels, the cost of keeping one warm standby instance overnight is significantly less than the SLO violation cost of a 3-minute cold start during an unexpected traffic event. Size the minimum based on your SLO sensitivity and your traffic unpredictability, not on the instinct to minimize idle GPU cost at all times.

9.3 Production Monitoring: Validating Your Sizing

Step 13 covers the initial validation. What follows is the ongoing monitoring framework - the metrics that tell you whether your fleet remains correctly sized as traffic evolves, and what each signal means for your next optimization decision.

Metric	Target	What it tells you
GPU memory utilization	80-90%	Below 70% - over-provisioned on memory. Above 95% - OOM risk on traffic spikes.
GPU compute utilization	60-85% during active serving	Consistently low during active requests - batch size too small or decode-dominated; batching improvement opportunity. Pegged at 100% with stable TTFT - prefill-bound but healthy. Pegged at 100% with rising TTFT - prefill-bound and saturated; consider chunked prefill or P/D disaggregation.
Adapter cache hit rate (if Multi-LoRA)	Above 80% for stable traffic	Below 60% - too many simultaneous adapter variants competing for GPU memory; increase `max_loras` budget or reduce adapter rank.
Adapter eviction rate (Multi-LoRA)	Near zero	Rising eviction rate - adapters are being paged to CPU and reloaded on demand, adding hidden latency. Symptom: TTFT spikes on specific tenants that look like prefill interference but do not respond to chunked prefill tuning.
KV cache utilization	75-90%	Near 100% - KV cache approaching ceiling; evaluate quantization or offloading before it becomes the bottleneck.
KV cache eviction rate	Below 5%	5–20% warning - KV budget undersized for current traffic. Above 20% - system in degraded state,

		presenting as a latency problem rather than a memory problem. This is the memory pressure cascade from Section 7 - instrument this metric from day one.
Fleet utilization	60-80%	Below 40% - you are in the utilization trap. Above 90% - no headroom for traffic spikes; size up or apply quantization to move the knee right.
Prefix cache hit rate	Workload-dependent	Below 30% on a workload with known prefix reuse - routing misconfiguration or cache too small. Single-instance deployments should hit 80%+ on RAG workloads with shared context.
TTFT P50 / P99	Within SLO	P99 violating but P50 fine - queue depth or prefill interference causing tail latency. Both violating - fleet undersized or workload has shifted beyond benchmark assumptions.
ITL / TPOT P50 / P99	Within SLO	P99 ITL spikes - prefill interference; try chunked prefill. Consistently elevated across all percentiles - decode batch too large or decode hardware undersized.
Queue depth	Near zero	Consistently growing - fleet undersized; requests are queueing faster than they are being served. Consistently zero at low GPU utilization - oversized fleet.
Tokens/GPU-hour	Maximize	Your core efficiency metric - reflects the combined effect of quantization, batching strategy, KV cache efficiency, and fleet utilization in one number.
Speculative decoding acceptance rate *(if enabled)*	Above 60%	Below 50% - draft model poorly matched to current traffic distribution; speculative decoding is adding overhead without benefit. Disable or retune before it becomes a latency liability.

Tokens/GPU-hour is the metric that connects everything. It is the single number that reflects your quantization choice, batching strategy, KV cache efficiency, and fleet utilization combined. If it is improving over time, your optimization work is paying off. If it is flat or declining as traffic grows, something is saturating - and the other metrics in this table will tell you what.

How to collect these metrics in practice. GPU memory utilization and compute utilization are available via nvidia-smi for single-node inspection and DCGM (Data Center GPU Manager) for fleet-level collection at scale - DCGM exposes Prometheus metrics that feed directly into Grafana dashboards. vLLM exposes a /metrics Prometheus endpoint natively covering TTFT, ITL, queue depth, KV cache utilization, prefix cache hit rate, and adapter eviction rate - enabling the entire monitoring table above with no custom instrumentation.

SGLang exposes equivalent metrics. For distributed tracing across prefill and decode nodes in disaggregated deployments, OpenTelemetry is the standard - trace context propagated from the prefill instance through the KV transfer to the decode instance lets you attribute latency to the correct phase rather than seeing an opaque end-to-end number.

Monitoring tells you whether your sizing was right. Guardrails determine what happens when traffic exceeds it.

9.4 Production Guardrails: Rate Limiting and Input Controls

Monitoring tells you when the fleet is stressed. Guardrails determine what happens to incoming traffic when it is. Without guardrails, a traffic spike that exceeds capacity does not produce clean 429 errors - it produces OOM cascades, runaway queue depth, and latency degradation that affects every active request simultaneously. The goal is to fail fast and cleanly at the edge rather than slowly and expensively inside the serving stack.

These are standard distributed systems patterns, but each has LLM-specific nuances worth making explicit.

Input token limits are the most LLM-specific guardrail. An unbounded input can consume the entire KV cache capacity of an instance in a single request, starving every other concurrent user. A model with a 128K context window set as the API limit will accept requests that consume 10 GB of KV cache per request at FP16 - exhausting an H100's entire KV cache budget in a single request at high concurrency. Your operational context limit and your model's architectural context limit are different numbers and must be set independently.

Set hard `max_input_tokens` limits at the API gateway layer - not inside the serving framework - so oversized requests are rejected before they consume GPU resources. The limit should reflect your P99 input length with reasonable headroom, not the model's maximum context length. For most production workloads, setting the operational limit at 2-3× your P95 input length provides enough headroom for legitimate long requests while preventing pathological cases from exhausting the KV cache.

Output token limits (`max_new_tokens`) prevent runaway generation from holding a decode slot indefinitely. A request generating 100K tokens occupies a decode slot for minutes - blocking other requests in disaggregated deployments and inflating queue depth in colocated ones. Set per-request output limits appropriate to your use case and enforce them at the gateway. For workloads where output length is genuinely unbounded - open-ended reasoning chains, agentic tasks - set a generous but finite limit and monitor P99 output length to detect drift.

Prompt injection and runaway generation detection is an operational guardrail that rate limiting alone does not catch. Adversarial or malformed inputs designed to force maximum output length - repetition loops, unbounded list generation, recursive reasoning prompts - can occupy decode slots for minutes and are not prevented by per-user rate limits if the user is within their RPS budget. In practice, detection at the gateway works through two complementary mechanisms. First, rule-based pattern matching on the input - regex or

keyword detection for known runaway patterns like "repeat X N times," "list all," or recursive self-reference constructs - catches the most common cases at near-zero latency cost. Second, a small output length predictor - typically a fine-tuned classifier on your historical request distribution that predicts whether a given input is likely to produce a long output - can flag statistically anomalous requests for either rejection or rerouting to a dedicated long-generation pool where they cannot starve interactive traffic. Neither mechanism is perfect. The rule-based approach misses novel patterns; the predictor adds 5-20ms of gateway latency and requires training data. For most teams, rule-based detection alone catches enough pathological cases to be worth the implementation cost, with the predictor added only if runaway generation becomes a measurable production problem.

Per-tenant rate limiting at the RPS level prevents a single tenant from monopolizing fleet capacity. Standard token bucket or leaky bucket algorithms apply - nothing LLM-specific beyond ensuring limits are enforced before requests enter the serving queue, not after. Enforcing inside the queue means the request has already consumed admission capacity and potentially KV cache budget before being rejected. Enforce at the gateway.

Circuit breaking - shedding load when queue depth exceeds a threshold rather than letting requests accumulate indefinitely - is critical for LLM fleets because queued requests can hold memory resources in some frameworks even before execution begins. A circuit breaker that returns HTTP 429 at queue depth above a threshold prevents a traffic spike from turning into an OOM cascade.

Set the threshold from your latency budget using Little's Law:

```
Maximum safe queue depth = maximum acceptable queue wait
time × arrival rate
```

For a fleet with a 2-second TTFT SLO and 50 RPS arrival rate, maximum queue depth before guaranteed SLO violation is 2 × 50 = 100 requests. Set your circuit breaker at 80–90% of that - 80 to 90 queued requests - to shed load before the guarantee is broken rather than after. Returning a 429 to 10% of requests during a spike is operationally preferable to returning degraded responses with 3× TTFT to 100% of requests.

9.5 Heterogeneous Fleet Composition

The sizing algorithm, monitoring framework, and guardrails in the previous sections all assume a homogeneous fleet. In practice, most mature deployments are not - and the heterogeneity is rarely planned. It accumulates across procurement cycles as infrastructure evolves faster than hardware generations.

Teams that deployed on A100s in 2022, added H100s in 2023, and are now evaluating MI300X for long-context workloads have three hardware generations in production simultaneously. This is not poor planning - it is the normal outcome of infrastructure that evolves faster than procurement cycles. The challenge is that each hardware tier requires its own benchmark curve, serving configuration, quantization artifact, and parallelism plan. You cannot treat heterogeneous nodes as interchangeable behind a load balancer.

The practical approach is to treat each hardware tier as an independent fleet tier - the same way model routing in Section 1 treats different model sizes as independent tiers. Route requests to the appropriate hardware based on their characteristics: long-context requests to MI300X where 192 GB HBM handles them without KV offloading, latency-sensitive interactive requests to H100s where NVLink and high TFLOPS minimize TTFT, batch summarization to A100s where throughput per dollar is the objective. Each tier runs its own optimized configuration - FP8 on H100s and Blackwell, INT8 on A100s, separate quantization artifacts per tier, separate benchmark curves, separate SLO-constrained operating points.

Three failure modes specific to heterogeneous fleets:

Mixed-tier load balancing without routing sends requests randomly across H100s and A100s. The result is a blended latency curve where the slower hardware tier sets the P99 for the entire fleet. A single A100 in a round-robin pool with seven H100s will receive roughly 12% of traffic - enough to make your P99 TTFT look like an A100 deployment even though 88% of your hardware is H100. Route by request characteristics, not randomly.

Assuming quantization format portability across hardware generations. FP8 Tensor Core acceleration is H100 and Blackwell-specific - an FP8 model artifact deployed on an A100 will run but will not use the dedicated hardware path, effectively running at INT8 performance without the INT8 memory footprint benefit. A100 deployments need INT8 quantization artifacts built and validated specifically for that hardware. One artifact does not serve both correctly.

Ignoring cold start asymmetry across hardware tiers. Model load time - the time to transfer weights from NVMe into GPU HBM - scales with HBM capacity and PCIe bandwidth, both of which differ across GPU generations. An autoscaling rule calibrated on H100 cold start times of 90 seconds will systematically underestimate A100 warm-up time and cause SLO violations during scale-up events on that tier. Measure cold start time independently per tier and set autoscaling pre-warm lead times accordingly.

The monitoring implication. The monitoring table in Section 9 must be instantiated independently per hardware tier with tier-specific targets - not aggregated across tiers. An H100 tier running at 85% GPU utilization and an A100 tier running at 55% blend to a fleet average of approximately 70% that looks healthy in aggregate dashboards. The A100 tier is in the utilization trap. The H100 tier may be approaching the memory ceiling. Neither signal is visible in the blended number. Tag all metrics by hardware tier from day one and alert on tier-specific thresholds, not fleet-wide averages.

The key principle across all of this: heterogeneity is a routing problem, not a sizing problem. Once you have the routing layer correctly directing traffic to the appropriate tier, each tier becomes a standard homogeneous fleet - and the full sizing algorithm applies cleanly to each one independently.

First Principles, Last

LLM inference infrastructure is one of the few places in modern software engineering where the gap between "it works" and "it works efficiently" translates directly into millions of dollars a year. A fleet running at 40% utilization with naive batching and no KV cache optimization is not a failing system - it serves requests, passes health checks, and looks fine on dashboards. It is just paying 3× what it needs to.

Most of the levers in this guide are not new ideas. Quantization, batching, caching, parallelism - these concepts predate LLMs. What is new is the specific way they interact in autoregressive generation: the prefill/decode duality, the quadratic KV cache growth with context length, the memory-bandwidth bottleneck that makes batch size the primary throughput lever rather than clock speed. Understanding these interactions, sequencing the decisions correctly, and measuring at each step is where most production deployments leave significant efficiency on the table - not because the engineers are inexperienced, but because the system is genuinely complex and the interactions are non-obvious until you have seen them in production.

The field will keep moving fast. New model architectures will change the memory math - Mixture-of-Experts already did, and whatever comes next will too. New hardware generations will shift the roofline ridge point and reopen hardware selection decisions that felt settled. New serving patterns will emerge the way disaggregated prefill-decode did - from a research paper to a production default in under two years. Specific numbers in this guide will age: GPU specs, framework defaults, benchmark results. Some already have between when sections were drafted and when you are reading this.

What does not age is the reasoning framework. The roofline model will correctly characterize any GPU workload regardless of the hardware generation. The four-component memory formula will correctly bound any transformer model regardless of its architecture. Little's Law will correctly describe any queuing system regardless of the serving framework. The latency-throughput curve will have a knee on any finite hardware system regardless of how much the knee moves rightward with each optimization. These are not LLM-specific insights - they are physics and queueing theory applied to a specific domain. They will outlast every framework version and hardware generation covered in this guide.

This guide is a starting point, not a finishing line. What stays constant is the value of reasoning from first principles: characterize the workload, understand the constraints, measure empirically, and size from data. Hopefully this guide makes that a little more principled.

ABOUT THE AUTHOR

VINAY JAYANNA

Vinay Jayanna is a Staff ML Engineer working on LLM inference optimization and Generative AI platform at a large-scale AI Platform. Previously at AWS, he led SageMaker's AI inference infrastructure serving production ML workloads at scale. He also founded Vipas.AI, an AI inference marketplace commercially hosting and serving large-scale LLMs. This guide reflects hands-on experience building and operating LLM infrastructure at scale - from the early days of cloud ML platforms to the current generation of LLM serving systems.

linkedin.com/in/vinayjayanna/

v1.0 · April 2026