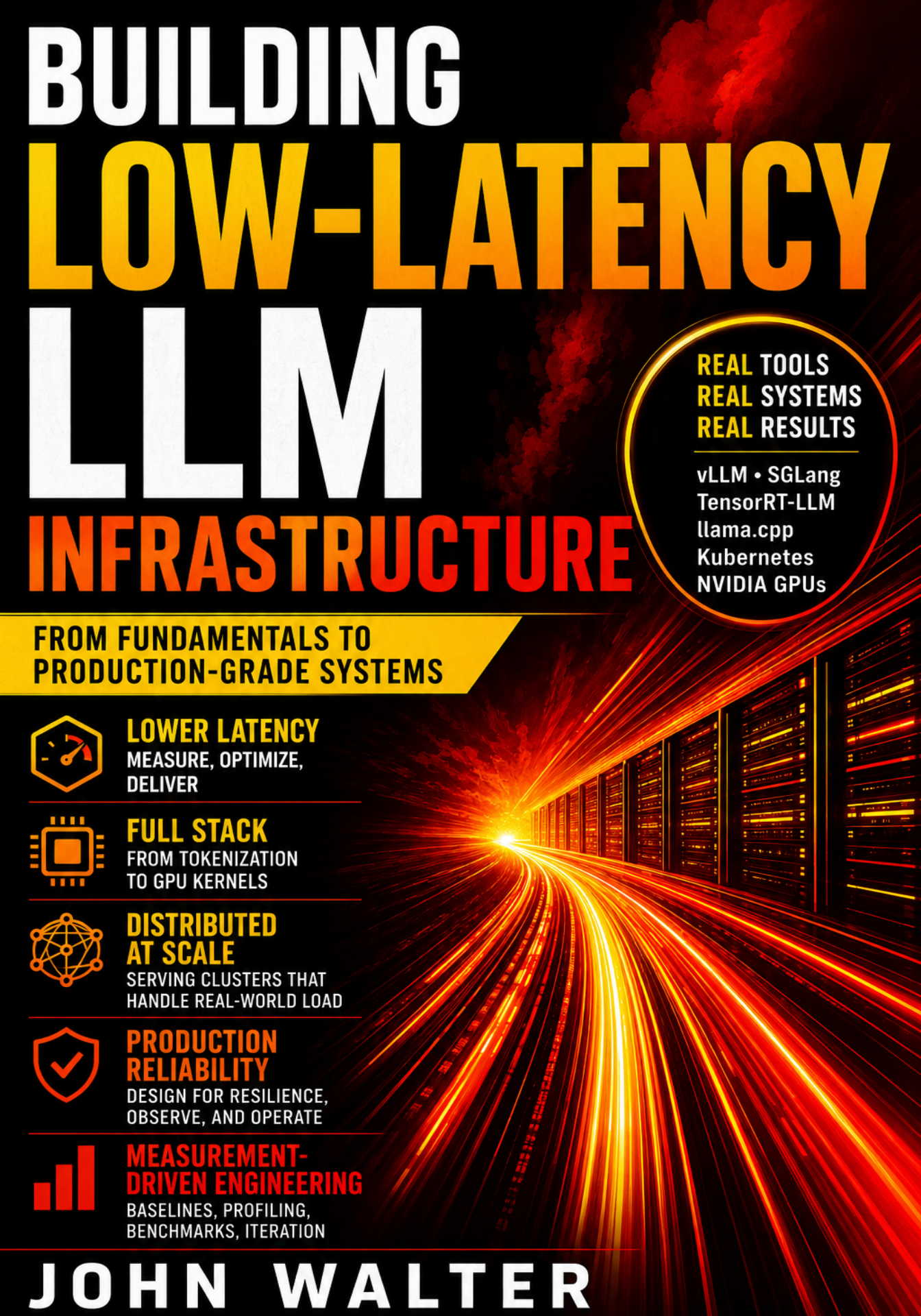
BUILDING LOW-LATENCY LLM INFRASTRUCTURE

REAL TOOLS
REAL SYSTEMS
REAL RESULTS

vLLM • SGLang
TensorRT-LLM
llama.cpp
Kubernetes
NVIDIA GPUs

FROM FUNDAMENTALS TO PRODUCTION-GRADE SYSTEMS



LOWER LATENCY
MEASURE, OPTIMIZE, DELIVER

FULL STACK
FROM TOKENIZATION TO GPU KERNELS

DISTRIBUTED AT SCALE
SERVING CLUSTERS THAT HANDLE REAL-WORLD LOAD

PRODUCTION RELIABILITY
DESIGN FOR RESILIENCE, OBSERVE, AND OPERATE

MEASUREMENT-DRIVEN ENGINEERING
BASELINES, PROFILING, BENCHMARKS, ITERATION

JOHN WALTER

Building Low-Latency LLM Infrastructure

From Fundamentals to Production-Grade Systems

Steve Publications

This book is available at
https://leanpub.com/buildinglow-latencyllminfrastructure

This version was published on 2026-08-18

Contents

From Fundamentals to Production-Grade Systems

This book teaches you how to design, build and operate large language model inference systems that deliver fast responses under real-world load. You will learn the full stack from tokenization and GPU kernels through distributed serving clusters and production reliability engineering. Every concept is explained intuitively first, then technically, then demonstrated with concrete implementation guidance using modern tools like vLLM, SGLang, TensorRT-LLM, llama.cpp, Kubernetes and NVIDIA GPUs. The book is organized as a progressive journey from understanding how a single request flows through an LLM to architecting geographically distributed platforms serving millions of requests. Measurement-driven engineering runs throughout: you will learn to define latency targets, establish baselines, profile bottlenecks, change one variable at a time, benchmark rigorously and iterate based on evidence rather than assumptions.

Introduction

A user types a question into a chat interface and presses Enter. How long until the first word appears? In the best systems today, less than half a second. In poorly designed ones, five seconds or more. That gap is not determined by the model itself but by everything surrounding it: how tokens are batched, how memory is managed on the GPU, whether requests queue behind each other, how many machines share the load, and what happens when one of them fails.

This book is about closing that gap systematically.

Large language models have made inference a first-class engineering problem. Training is expensive but periodic; inference is continuous and scales directly with users. A model that runs slowly at ten requests per second becomes unusable at a thousand. The techniques that work for a development prototype collapse under production load: naive batching starves interactive users, unmanaged memory fills the GPU in minutes, and tail latency spikes turn acceptable average performance into frustrated customers.

The central thesis of this book is straightforward: low-latency LLM inference is not about any single optimization trick. It is a full-stack engineering discipline where you must understand and allocate latency budgets across tokenization, networking, scheduling, memory management, kernel execution and inter-node communication. The optimal architecture depends on precise workload characterization, measured bottlenecks and explicit trade-off decisions between latency, throughput, cost and reliability. There are no universal best choices; there are only the right choices for your specific constraints.

Who This Book Is For

This book assumes you can write code in Python or a similar language and understand basic concepts like APIs, containers and databases. You do not need prior experience with machine learning systems, CUDA programming or distributed computing. The material progresses from fundamentals through advanced topics, so you can read it sequentially as a complete curriculum or jump to specific sections as reference.

The intended readers fall into several overlapping categories:

- **ML engineers** who have trained or fine-tuned models and now need to serve them reliably at scale
- **Platform and infrastructure engineers** building the systems that host inference workloads
- **Software engineers** integrating LLMs into products who want to understand performance characteristics and architectural trade-offs
- **Experienced practitioners** looking for a systematic reference on advanced techniques like disaggregated architectures, speculative decoding, or topology-aware scheduling

How This Book Is Organized

The book is structured as a learning progression that mirrors how you would actually build an inference system in practice.

Part I covers the fundamentals of LLM inference: what happens when a request arrives, the two distinct phases of processing (prefill and decode), the key-value cache, and the latency metrics that matter most to users and engineers. Without this foundation, optimization is guesswork.

Part II dives into hardware: GPU architecture from an inference perspective, the CPU-GPU interface, and why certain operations are fast while others are painfully slow. You do not need to become a CUDA expert but you must understand what the hardware is doing under the hood.

Part III covers the core optimization techniques that every production system uses: batching strategies, KV cache management, quantization and advanced decoding methods like speculative decoding. These are the levers you will pull first when optimizing any inference workload.

Part IV addresses distributed inference for models that do not fit on one GPU or workloads that require more throughput than a single machine can provide: tensor parallelism, pipeline parallelism, interconnects and fault tolerance.

Part V covers serving architecture from building a single inference server to operating cluster-scale systems with load balancing, autoscaling, admission control, and multi-model deployments.

Part VI focuses on production engineering: observability, reliability, capacity planning, cost optimization, and the operational practices that keep systems running smoothly at scale.

Part VII explores advanced topics for specialized workloads: extremely low-latency serving, real-time voice assistants, long-context inference, mixture-of-experts models, and emerging techniques from research.

Part VIII brings everything together with four complete reference architectures at increasing scales, each fully specified with deployment steps, expected bottlenecks, and capacity planning methodology. The final part synthesizes the book into a practical framework for designing new systems from first principles.

A Note on Tools and Versioning

The LLM infrastructure landscape changes rapidly. Frameworks like vLLM and SGLang ship major updates every few months. NVIDIA releases new GPU architectures annually. Cloud providers adjust pricing quarterly. This book separates universal architectural principles from tool-specific implementations wherever possible, but it necessarily references specific tools and versions to be concrete and actionable.

When performance numbers or feature availability are stated, they reflect the state of the art as of early 2026. Always verify critical details against current documentation before making production decisions. The principles will outlast any particular tool version; the specifics may not.

How to Read This Book

Read sequentially if you are new to LLM infrastructure. Each chapter builds on concepts introduced earlier, and jumping ahead will leave gaps in your understanding of why certain design choices exist.

If you are experienced and need specific information, use the chapter and section headings to navigate directly to relevant topics. The reference architectures in Part VIII assume familiarity with earlier material but can serve as standalone deployment guides once you understand the underlying concepts.

Work through the examples when possible. Understanding KV cache math on paper is different from calculating it for your actual model and realizing you cannot fit three concurrent users. Running a benchmark before and after enabling continuous batching teaches more than any description of the algorithm.

The single most important habit this book tries to instill is measurement before optimization. Define what good looks like, measure where you are now, profile to find the bottleneck, change one thing, measure again. Repeat. Everything else follows from that discipline.

Chapter 1: The Anatomy of a Single Request

Before optimizing anything, you must understand what happens when a request travels through an inference system. This chapter walks through the complete lifecycle of a single LLM inference request from the moment it leaves a client application to the moment the final token arrives back at that client. Every optimization discussed later in this book targets one or more stages described here.

From HTTP Request to Generated Token: The Complete Lifecycle

A typical production inference request passes through many components before any computation on the model occurs, and many more after. Consider a user typing "Explain quantum entanglement in simple terms" into a web application that calls an internal LLM service.

The journey begins at the client. The browser or mobile app serializes the prompt into an HTTP request, usually following the OpenAI-compatible API format with JSON payload containing the model name, prompt text, temperature and other parameters. This request travels across the public internet to a load balancer or API gateway that terminates TLS, authenticates the caller, validates the request format and forwards it to one of many backend inference servers.

The inference server receives the request and performs several operations before touching the GPU. It tokenizes the input text into numerical IDs using the model's tokenizer. It checks whether the current batch has room for this request or whether it must wait in a queue. If admitted, the request joins the batch and enters the prefill phase where the model processes all input tokens in parallel to build the key-value cache.

Once prefill completes, the first generated token is ready. The server serializes it into the response format and sends it back through the load

balancer to the client. This moment marks Time-to-First-Token (TTFT), one of the most critical latency metrics for user experience. But the work is not done. The model now enters the decode phase, generating tokens one at a time in an autoregressive loop. Each new token becomes input for the next prediction until the model outputs an end-of-sequence token or reaches the maximum length limit.

During decode, each generated token is streamed back to the client as soon as it is ready, typically using Server-Sent Events (SSE) over a persistent HTTP connection. The user sees text appearing word by word rather than waiting for the complete response. After all tokens are generated and delivered, the request lifecycle ends.

Every stage in this flow adds latency: network round trips between client and datacenter, processing time at the API gateway, tokenization overhead, queueing delay if the server is busy, prefill computation, decode steps, serialization and response transmission. Optimizing LLM inference means understanding where each millisecond goes and deciding which stages are worth optimizing given your workload characteristics.

Tokenization: The Hidden Cost Before Computation Begins

Tokenization converts raw text into sequences of integer IDs that the model can process. This step happens before any GPU computation and is frequently overlooked in latency budgets, but it can consume tens to hundreds of milliseconds depending on prompt length and implementation.

Modern LLMs use subword tokenizers based on algorithms like Byte Pair Encoding (BPE) or SentencePiece. These tokenizers split text into units that balance vocabulary size against fragmentation: common words like “the” become single tokens while rare words like “quantum” might be split into multiple subword tokens like “quant” and “##um”. The exact vocabulary and splitting rules are determined during model training and cannot be changed without retraining.

The tokenizer maintains a vocabulary mapping from token strings to integer IDs, typically containing 32,000 to 150,000 entries for modern models. Encoding text requires scanning the input and applying the learned merge operations,

which is computationally nontrivial for long prompts. Decoding token IDs back to text is simpler but still requires dictionary lookups for each token.

Tokenization overhead scales with input length. A short prompt of 10 words might tokenize in under 5 milliseconds on a modern CPU. A document paste of 50,000 tokens can take 200 milliseconds or more depending on the implementation and whether Unicode normalization is required. For latency-sensitive applications processing long contexts, tokenization can become a significant portion of TTFT.

Several factors affect tokenization performance in production:

- **Implementation language**: Python-based tokenizers (like Hugging Face's `transformers` library) are slower than C++ implementations (like those shipped with vLLM or llama.cpp). The difference matters at scale where thousands of requests per second each require encoding.
- **Unicode handling**: Text containing emojis, non-Latin scripts, or mixed-direction content requires normalization that adds overhead. Some tokenizers handle this more efficiently than others.
- **Caching opportunities**: In multi-turn conversations with fixed system prompts, the system prompt tokens can be cached and reused across requests, eliminating redundant encoding work.

A practical example illustrates the impact. Consider a RAG application where each request includes a 4,000-token document context plus a user question. At 100 requests per second, you are tokenizing 400,000 tokens per second continuously. If your tokenizer processes at 500,000 tokens/second on available CPU cores, tokenization alone consumes 20 percent of your compute capacity before the GPU is involved.

Prefill: Processing the Input Prompt in One Pass

The prefill phase processes all input tokens through the complete transformer stack in a single forward pass. This is where the model reads and understands your prompt before generating any output.

During prefill, the model takes the full sequence of input token IDs and computes attention scores between every pair of tokens. For each layer of the transformer, it also computes feed-forward network activations. The result is a

set of key and value tensors for each token at each layer, stored in what is called the KV cache (key-value cache). These cached values allow subsequent decode steps to avoid recomputing attention over the entire prompt from scratch.

Prefill has two defining characteristics that make it fundamentally different from decoding:

First, prefill is highly parallelizable. All input tokens are known at the start and can be processed simultaneously through matrix-matrix multiplications. This means the GPU's tensor cores work at full capacity performing large batched operations. Prefill is typically compute-bound: limited by how fast the GPU can perform arithmetic rather than how fast it can move data between memory and compute units.

Second, prefill duration scales with input length. Processing a 50-token prompt takes roughly fifty times less time than processing a 2,500-token prompt, all else being equal. For models with quadratic attention complexity (standard self-attention), the scaling is actually worse: doubling the sequence length quadruples the attention computation. Modern optimizations like FlashAttention reduce this to approximately linear scaling in practice but the dependence on input length remains strong.

The prefill phase dominates Time-to-First-Token for long prompts. If a user submits a 10,000-token document for summarization, most of the wait time before seeing any output is spent in prefill building the KV cache. This has direct implications for system design: you cannot optimize TTFT for long-context workloads by speeding up decode alone.

Batching during prefill improves throughput significantly because the same model weights are reused across multiple prompts processed together. A batch of ten 500-token prompts can be processed almost as fast as a single 5,000-token prompt since the heavy matrix operations scale favorably with batch dimension. However, mixing very long and very short prompts in the same prefill batch creates imbalance: shorter requests wait for longer ones to complete before they can start decoding.

Decoding: Generating Tokens One at a Time

After prefill builds the KV cache, the model enters the decode phase where it generates output tokens one at a time in an autoregressive loop. This is the heart of LLM generation and also its fundamental bottleneck.

Autoregressive generation works as follows. The model takes the last generated token (or the last input token for the first output token) and computes a probability distribution over all possible next tokens from the vocabulary. A sampling method selects one token from this distribution based on parameters like temperature and top-p. That selected token becomes both part of the output and the input for the next prediction step. The process repeats until an end-of-sequence token is generated or a maximum length limit is reached.

This sequential dependency means decoding cannot be parallelized across tokens in the straightforward way that prefill can. Each token must wait for its predecessor. If generating each token takes 50 milliseconds, producing a 100-token response requires at least 5 seconds of decode time regardless of how many GPUs you throw at it (within the constraints of techniques discussed later like speculative decoding).

More importantly for system design, decoding is typically memory-bandwidth-bound rather than compute-bound. At each decode step, the model must read all its weights from GPU memory to perform the forward pass, but it only processes a single token's worth of new input. The ratio of computation performed to data moved (called arithmetic intensity) is very low: approximately 1 FLOP per byte for the linear layers that dominate transformer computation. Modern GPUs have far more compute capacity than memory bandwidth, so they spend most of their time waiting for data rather than performing calculations.

This memory-bound characteristic has several consequences:

- **Batching helps but with diminishing returns**: Processing multiple requests simultaneously during decode improves memory bandwidth utilization because the same weights are shared across all batched tokens. However, increasing batch size also increases KV cache reads (which grow with total sequence length), eventually saturating memory bandwidth without further throughput gains.
- **Memory bandwidth is king**: GPUs with higher memory bandwidth (H100 with 3.35 TB/s HBM3 versus A100 with 2.04 TB/s HBM2e) decode noticeably faster for the same model, often proportionally to the bandwidth ratio.
- **KV cache size directly affects latency**: As sequences grow during generation, more KV cache data must be read at each step, consuming bandwidth that could otherwise load model weights.

The decode phase determines Time-Per-Output-Token (TPOT), the average latency between consecutive generated tokens. TPOT of 50 milliseconds means users see about 20 tokens per second, which feels like fast but natural reading speed. TPOT above 200 milliseconds (fewer than 5 tokens per second) feels sluggish and causes users to question whether the system is still working.

The Key-Value Cache: Trading Memory for Recomputation

The KV cache is perhaps the single most important data structure in LLM serving optimization. Understanding it is essential because every major inference optimization technique either manages the KV cache more efficiently or reduces its memory footprint.

During attention computation, each token produces a query vector (Q), a key vector (K), and a value vector (V). The query from the current token is compared against all previous keys to compute attention weights, which are then applied to the values to produce the attention output. Without caching, each new decode step would need to recompute keys and values for all previously processed tokens, leading to quadratic computation growth as the sequence lengthens.

KV caching avoids this recomputation by storing the key and value vectors for every token after they are first computed. When generating a new token, the model only computes QK and V for that single new token, then attends over the cached keys and values from all previous tokens plus the new ones. This reduces attention computation from quadratic to linear in sequence length during decoding.

The trade-off is memory. The KV cache grows linearly with both sequence length and batch size, and its total size can quickly exceed the model weights themselves for long contexts or large batches.

Here is the exact formula for KV cache memory usage:

KV_cache_bytes = sequence_length × num_layers × 2 × num_kv_heads × head_dim × dtype_bytes × batch_size

Breaking this down:

- `sequence_length` is the total number of tokens (prompt plus generated)

- `num_layers` is the number of transformer layers in the model
- The factor of 2 accounts for both keys and values
- `num_kv_heads` is the number of key-value attention heads (may differ from query heads in GQA/MQA architectures)
- `head_dim` is the dimensionality of each attention head
- `dtype_bytes` is 2 for FP16/BF16, 4 for FP32, 1 for INT8
- `batch_size` is the number of concurrent sequences

Concrete examples make the scale clear. For Llama 3 8B with 32 layers, 8 KV heads (using GQA), and head dimension 128:

A single request with 4,096 tokens at FP16 uses approximately 52 MB of KV cache. Ten concurrent requests use 520 MB. Now consider Llama 3 70B with 80 layers, 8 KV heads, and head dimension 256: a single request at 128K context length requires roughly 42 GB of KV cache alone, leaving almost no room for model weights on an 80 GB GPU.

This memory pressure drives virtually every KV cache optimization discussed in this book: PagedAttention eliminates fragmentation to pack more requests into available memory; prefix caching shares cached blocks across requests with common prompts; quantization reduces dtype_bytes from 2 to 1 or less; and eviction policies decide which cached tokens to discard when memory runs out.

Managing the KV cache efficiently is not an optional optimization for production systems: it is the central challenge that determines how many users you can serve simultaneously on a given GPU, how long contexts you can support, and whether your system crashes with out-of-memory errors under load.

Latency Budgets: TTFT, TPOT, and What They Mean for Users

Latency in LLM inference is not a single number. Different phases of the request lifecycle contribute to different aspects of user experience, and optimizing one metric often trades off against another. Production systems track several latency metrics, each answering a distinct question about performance.

Time-to-First-Token (TTFT) measures the elapsed time from when a client submits a request until the first output token arrives. TTFT captures everything

before generation begins: network latency to reach the server, API gateway processing, tokenization, queueing delay if the server is busy, and the entire prefill phase. For interactive applications like chat interfaces, TTFT determines whether users perceive the system as responsive. Research on user experience suggests that sub-500 millisecond TTFT feels instant, 500ms to 1 second feels acceptable, and anything over 2 seconds creates noticeable friction where users wonder if they need to retry.

Time-Per-Output-Token (TPOT) measures the average latency between consecutive generated tokens during the decode phase. TPOT determines how smoothly text streams to the user after the first token appears. A TPOT of 50 milliseconds delivers about 20 tokens per second, which matches natural reading speed for most users and creates the illusion that the model is "thinking aloud." TPOT above 150 milliseconds (below 7 tokens per second) feels slow enough to distract users from the content.

Inter-token latency variance describes how consistent the time between tokens is. Even with good average TPOT, large variance causes stuttering where tokens arrive in bursts followed by pauses. Users perceive this as lag or network problems even when total latency is acceptable. Variance typically arises from scheduling interference (a long prefill request joining the batch mid-decode), garbage collection pauses, or variable computation across different model layers.

End-to-end latency is the total time from request submission until the complete response is delivered. For short responses, end-to-end latency is dominated by TTFT. For long responses with hundreds of generated tokens, decode time dominates and end-to-end latency can be many seconds even with excellent TTFT and TPOT individually.

These metrics are interconnected through queueing dynamics. When a server is lightly loaded, requests experience minimal queueing delay and both TTFT and TPOT approach their theoretical minimums. As load increases and the server approaches capacity, requests wait longer in queues (increasing TTFT) and batches grow larger to maximize utilization (which can increase TPOT due to memory bandwidth contention). The relationship is nonlinear: small increases in utilization near capacity cause disproportionately large increases in tail latency, a phenomenon described by queueing theory and explored in detail in Chapter 2.

Different workloads prioritize different metrics. A real-time voice assistant

needs extremely low TTFT (under 200 milliseconds) because silence feels like the system is broken. A batch summarization job processing thousands of documents can tolerate high TTFT if overall throughput is maximized. A code completion tool needs both low TTFT and low TPOT variance so suggestions appear smoothly as developers type. Understanding which metrics matter for your specific use case determines where to focus optimization effort.

Summary

A single LLM inference request passes through tokenization, prefill, decode and response streaming, with each phase having distinct performance characteristics. Tokenization happens on the CPU and scales with input length. Prefill is compute-bound and parallelizable, dominating TTFT for long prompts. Decode is memory-bandwidth-bound and sequential, determining TPOT and total generation time. The KV cache eliminates redundant computation at the cost of memory that grows linearly with sequence length and batch size. Latency metrics like TTFT and TPOT capture different aspects of user experience and often conflict when optimized simultaneously.

The next chapter examines how to measure these characteristics rigorously, establish baselines, and use metrics to drive optimization decisions rather than guesswork.

Chapter 2: Measuring What Matters

You cannot optimize what you do not measure. This statement is so common it has become cliché, but in LLM inference systems it is worth repeating because the stakes are high and the traps are numerous. A system that shows excellent average latency in a quick test can deliver unusable tail latency under real load. A GPU reporting 95 percent utilization might be completely memory-bound with compute cores sitting idle. Benchmarks run on synthetic data often tell a different story than production traffic.

This chapter establishes the measurement framework that drives every optimization decision in this book. You will learn which metrics actually inform engineering decisions, how queueing theory predicts system behavior under load, what profiling tools reveal about bottlenecks, and how to design benchmarks that produce trustworthy results.

Latency Metrics That Actually Inform Decisions

The first rule of latency measurement is that averages lie. Reporting "our P50 TTFT is 200ms" while your P99 is 8 seconds tells you almost nothing useful about user experience because the users who matter most are the ones hitting the tail, not the median.

Latency distributions in inference systems are typically right-skewed: most requests complete quickly but a small fraction take much longer due to queueing delays, scheduling interference, memory pressure, or hardware variability. This skew means that as you optimize for average latency, tail latency can remain unchanged or even worsen if your optimization increases variance.

Production systems should track at minimum:

- **P50 (median)**: Where half of requests land. Useful for understanding typical behavior but insufficient alone.
- **P95**: The latency below which 95 percent of requests complete. A reasonable proxy for "most users most of the time."

- **P99**: The latency below which 99 percent of requests complete. This is where user-facing SLAs are typically defined because it captures the experience of unlucky but not pathological requests.
- **P99.9**: Relevant for very high-traffic systems where even 0.1 percent of bad requests affects thousands of users per hour.

The gap between these percentiles tells you about variance. If P50 is 200ms and P99 is 400ms, your system has low variance and predictable performance. If P50 is 200ms and P99 is 5 seconds, something is causing occasional severe slowdowns that need investigation: perhaps a long prefill request is blocking the batch, or memory pressure triggers eviction, or there are periodic GC pauses.

How you measure latency matters as much as what you measure. Common mistakes include:

- **Measuring server-side only**: Server-side TTFT excludes network latency between client and server. For geographically distributed users, network RTT can dominate perceived latency even if the server is fast.
- **Including warmup in measurements**: The first few requests after model load are slower due to GPU warmup and kernel compilation. Always discard an initial warmup period before collecting metrics.
- **Running benchmarks without concurrent load**: A single isolated request tells you nothing about behavior under realistic concurrency where queueing and batching effects appear.
- **Using synthetic prompts that do not match production**: Short random prompts produce different performance than long documents or structured code input. Use production-like data distributions.

A practical approach is to instrument your system to record latency for every request with sufficient metadata: model name, prompt length, output length, batch size at admission, queue wait time, and timestamp. Store these records in a time-series database or logging system that supports percentile computation over sliding windows. This allows you to answer questions like "what was our P99 TTFT for requests with prompts longer than 4,000 tokens during the last hour" rather than relying on aggregate numbers that obscure important patterns.

Throughput vs Latency: The Fundamental Trade-off

Throughput and latency are not independent knobs you can turn separately. In LLM inference systems, they exist in a fundamental trade-off governed by how you batch requests and manage resources.

Throughput measures how much work the system completes per unit time, typically expressed as tokens per second or requests per second. Latency measures how long individual requests take, expressed as TTFT, TPOT, or end-to-end duration. Increasing throughput generally requires higher resource utilization, which increases contention and degrades latency for individual requests.

The relationship manifests clearly in batching decisions. Consider a GPU serving Llama 3 8B with decode phase memory bandwidth of approximately 1.5 TB/s utilized at 80 percent efficiency:

- Batch size 1: The GPU processes one token at a time, reading all model weights for each step. Throughput might be 40 tokens per second with TPOT of 25ms.
- Batch size 8: The same weight reads serve eight tokens simultaneously. Throughput might increase to 200 tokens per second but TPOT rises to 60ms because memory bandwidth is shared across eight sequences and KV cache reads have grown.
- Batch size 32: Throughput peaks around 400 tokens per second but TPOT degrades to 150ms as memory bandwidth saturates and variance between sequence lengths causes inefficiency.

The curve is not linear and depends heavily on model size, GPU characteristics, and workload patterns. Small models on fast GPUs reach their compute limit at lower batch sizes than large models that remain memory-bound. Workloads with uniform output lengths batch more efficiently than workloads where some requests generate 10 tokens and others generate 500.

This trade-off forces explicit prioritization. Systems optimized for interactive latency keep batch sizes small enough to maintain TPOT below a target threshold (say 80ms), accepting lower total throughput. Systems optimized for batch processing run large batches maximizing tokens per second, accepting that individual requests may take longer to complete.

There is no universal optimal point. The right balance depends on your workload:

- **Interactive chat**: Prioritize low TTFT and TPOT over raw throughput. Users abandon slow conversations.
- **Batch document processing**: Prioritize throughput. Requests can queue and process asynchronously.
- **Mixed workloads**: Require more sophisticated scheduling that segregates latency-sensitive and throughput-oriented traffic into different queues or even different server pools.

Utilization Metrics: GPU Compute and Memory Bandwidth

GPU utilization is commonly reported as a percentage but the number alone is meaningless without understanding what it measures and what it does not.

Modern NVIDIA GPUs expose several utilization counters through tools like `nvidia-smi` and DCGM:

- **SM (Streaming Multiprocessor) utilization**: The fraction of time GPU compute cores are executing instructions. High SM utilization suggests the workload is using available compute capacity.
- **Memory utilization**: The fraction of time memory controllers are active transferring data. High memory utilization suggests the workload is limited by how fast it can move data between HBM and compute units.
- **Tensor core utilization**: Specific to matrix multiplication operations using tensor cores. Relevant for transformer workloads that heavily use GEMM operations.

For LLM inference, the most informative observation comes from comparing SM utilization against memory utilization:

- **High SM, low memory utilization**: The workload is compute-bound. You are limited by arithmetic capacity and might benefit from faster GPUs with more tensor cores or from algorithmic optimizations that reduce computation (quantization, early exit).

- **Low SM, high memory utilization**: The workload is memory-bound. This is typical for decode phase of LLM inference. You are limited by memory bandwidth and should consider GPUs with higher bandwidth (H100 over A100), quantization to reduce data movement, or increasing batch size to improve arithmetic intensity.
- **Both low**: The GPU is underutilized due to insufficient work. Increase concurrency, check for CPU-side bottlenecks like slow tokenization or serialization, or investigate kernel launch overhead.

A common misconception is that high utilization always indicates good performance. For latency-critical workloads, 100 percent utilization often means the system is fully loaded with no headroom for burst traffic, leading to queueing delays and tail latency spikes. Production systems targeting sub-second TTFT typically operate at 60 to 80 percent average utilization to maintain responsiveness under variable load.

Another subtlety: utilization measured over long time windows (one minute or more) smooths out important short-term patterns. A GPU might show 70 percent average utilization while actually running at 100 percent for 45 seconds followed by 20 percent idle time as batches complete and new requests arrive. Shorter sampling intervals (1 to 5 seconds) reveal these dynamics and help diagnose scheduling issues.

Queueing Theory for Inference Systems

Queueing theory provides mathematical models that predict how latency, throughput and utilization interact in systems where work arrives randomly and must wait for service. While real inference systems are more complex than textbook queueing models, the basic principles offer powerful intuition for capacity planning and understanding why tail latency explodes near capacity.

The simplest useful model is M/M/1: a single server with Poisson-distributed arrivals and exponentially distributed service times. The key variable is utilization □ (rho), defined as arrival rate □ divided by service rate □. When □ approaches 1, the system becomes saturated and queue lengths grow dramatically.

For an M/M/1 queue, the average waiting time in queue (before service begins) is:

$$W_q = \rho / (\mu \times (\mu - \lambda))$$

Notice what happens as λ approaches μ. If your server can handle 100 requests per second ($\mu = 100$) and you receive 90 requests per second ($\lambda = 90$), utilization is 90 percent and average queue wait is 0.9 seconds. Increase arrival rate to 99 requests per second and queue wait jumps to 9.9 seconds despite only a 10 percent increase in load. At 99.9 percent utilization, average queue wait is nearly 100 seconds.

This nonlinear relationship explains why production systems must operate with headroom. Running at 95 percent utilization might seem efficient but produces unpredictable tail latency that violates SLAs during normal traffic variation. A common rule of thumb for latency-sensitive services is to size capacity so that expected peak load produces utilization no higher than 70 to 80 percent.

LLM inference systems deviate from M/M/1 in several ways that make them both better and worse:

- **Better**: Batching means one GPU service operation handles multiple requests simultaneously, effectively increasing service rate μ as queue length grows (up to the point of memory saturation).
- **Worse**: Service times are highly variable. A request generating 500 tokens takes ten times longer than one generating 50 tokens, and this variation is not captured by simple exponential distributions. This variability increases tail latency beyond M/M/1 predictions.
- **Worse**: Requests interfere with each other through shared resources. A long prefill request joining the batch can stall decode for all other requests in that batch, creating bursty latency patterns not predicted by independent service time models.

Despite these deviations, queueing theory provides essential guidance:

- Never design a system where expected load approaches theoretical maximum capacity
- Monitor queue length as an early warning signal: growing queues precede latency spikes
- Use Little's Law ($L = \lambda W$, where L is average number of items in system, λ is arrival rate, W is average time in system) to relate concurrency, throughput, and latency

A practical application: if you know your target P99 TTFT must be under 500ms and your prefill phase takes 200ms on average, you have 300ms budget for queueing delay. Using queueing models with your measured service time distribution, you can calculate the maximum arrival rate that keeps expected queue delay below this budget, then size your GPU fleet accordingly.

Profiling Fundamentals: Traces, Metrics, and Flame Graphs

When a system is slow, profiling tells you where time is actually spent rather than where you assume it is spent. Three complementary observability approaches cover different aspects of inference system behavior.

Distributed traces follow individual requests through the entire system, recording timestamps at each stage. For an LLM request, a trace might show: HTTP arrival at load balancer (0ms), forwarded to inference server (3ms), tokenization complete (15ms), queued for 45ms, prefill started (60ms), prefill complete (380ms), first decode step (420ms), second decode step (470ms) and so on. Traces are invaluable for diagnosing latency issues in specific requests: was the delay in queueing, prefill or decode? Did one request block others?

Metrics are time-series aggregates computed from many requests: request rate, error rate, P50/P95/P99 latency distributions, GPU utilization, queue length. Metrics answer questions about system-wide behavior over time: is performance degrading gradually? Did the last deployment cause a regression? Are we approaching capacity limits? Metrics alone cannot explain why a specific request was slow but they identify when investigation is needed.

Flame graphs and GPU profilers show where computation time is spent inside the model execution itself. Tools like NVIDIA Nsight Systems, Nsight Compute, and PyTorch's profiler capture kernel execution timelines, memory transfer durations, and CPU-GPU synchronization points. A flame graph might reveal that 60 percent of decode time is spent in attention kernels reading KV cache, 25 percent in feed-forward layers loading weights, and 15 percent in overhead (kernel launches, synchronization, Python interpreter). This level of detail guides kernel-level optimization decisions.

The relationship between these tools follows a funnel pattern:

1. Metrics alert you that something is wrong (P99 latency increased 50 percent)
2. Traces show which requests are affected and where in the pipeline delay occurs (prefill taking 3x longer than normal)
3. GPU profilers reveal the low-level cause (attention kernel spending excessive time on memory transfers due to fragmented KV cache)

Production systems should implement all three layers. OpenTelemetry provides a vendor-neutral framework for collecting traces and metrics that can be exported to backends like Jaeger, Tempo or commercial observability platforms. NVIDIA DCGM Exporter integrates GPU metrics into Prometheus ecosystems. For GPU profiling in production, lightweight continuous profiling tools like Nsight Systems in sampling mode provide visibility without the overhead of full trace collection on every request.

Building a Baseline Benchmark Harness

Before optimizing anything, you must establish a reproducible baseline that characterizes current system performance under controlled conditions. Ad hoc testing with curl commands or browser requests produces numbers you cannot trust or compare against future changes.

A proper benchmark harness controls and reports the following:

- **Workload definition**: The dataset of prompts used for testing, including distribution of prompt lengths, expected output lengths, and any structural characteristics (code, natural language, mixed). Use production traffic samples when available; otherwise use established datasets like ShareGPT or custom synthetic workloads that match your use case.
- **Load pattern**: How requests arrive over time: all at once (offline batch mode), constant rate (N requests per second), or Poisson-distributed inter-arrival times (more realistic for user traffic). Different load patterns produce different performance characteristics.
- **Warmup protocol**: Number of initial requests discarded to allow GPU warmup, kernel caching, and steady-state operation before measurement begins. Typical warmup: 10 to 50 requests depending on system size.
- **Measurement duration**: Long enough to collect statistically significant data (at least several hundred completed requests) but short enough to complete in reasonable time.

- **Metrics collected**: TTFT distribution, TPOT distribution, end-to-end latency, throughput (tokens/second and requests/second), GPU utilization, queue length over time, error rate.
- **Environment documentation**: Exact model version, framework version, GPU type and count, driver version, container image, configuration parameters. Without this information, benchmark results cannot be reproduced or compared.

Here is a minimal example of how to structure a benchmark using vLLM's built-in benchmarking tool:

```
# Baseline benchmark: Llama 3 8B on single A100, ShareGPT dataset
vllm bench serving \
    --backend vllm \
    --model meta-llama/Llama-3.1-8B-Instruct \
    --dataset-name sharegpt \
    --num-prompts 500 \
    --request-rate 10 \
    --max-model-len 4096 \
    --output-file baseline_llama8b_a100.json
```

This command sends 500 prompts from the ShareGPT dataset at a rate of 10 requests per second and records detailed metrics. The output JSON file contains latency percentiles, throughput numbers, and per-request breakdowns that you can compare against future optimization runs.

For more control over workload characteristics, use synthetic workloads with explicit prompt/output length distributions:

```
# Synthetic benchmark: long prompts, short outputs (RAG-like pattern)
vllm bench serving \
    --backend vllm \
    --model meta-llama/Llama-3.1-8B-Instruct \
    --dataset-name random \
    --num-prompts 1000 \
    --request-rate 5 \
    --random-input 4096 \
    --random-output 128 \
    --output-file rag_pattern_baseline.json
```

When comparing benchmark results, focus on relative changes rather than absolute numbers. A 20 percent improvement in throughput with no degradation in P99 TTFT is meaningful regardless of whether baseline throughput

was 100 or 1000 tokens per second. Always report confidence intervals when possible: run benchmarks multiple times and report mean plus/minus standard deviation to account for system noise.

Summary

Measurement drives optimization. Track latency percentiles (P50, P95, P99) rather than averages because tail latency determines user experience. Understand the trade-off between throughput and latency: you cannot maximize both simultaneously and must prioritize based on workload requirements. Monitor GPU utilization with attention to whether compute or memory is the limiting factor. Use queueing theory to understand why tail latency explodes near capacity and to size systems with appropriate headroom. Implement traces, metrics and profiling as complementary observability layers that work together to diagnose issues. Establish reproducible benchmark harnesses before optimizing so you can measure improvement rigorously rather than relying on anecdotal evidence.

The next chapter dives into GPU architecture from an inference perspective, explaining why certain operations are fast and others are slow at the hardware level.

Chapter 3: GPU Architecture for Inference

GPUs were originally designed for graphics rendering but have become the dominant hardware for deep learning because their architecture excels at the massively parallel matrix operations that dominate neural network computation. Understanding GPU architecture from an inference perspective does not require becoming a CUDA expert, but it does require understanding why certain patterns are fast and others are slow, how memory hierarchy affects performance, and what characteristics of LLM workloads interact with hardware limitations.

This chapter focuses exclusively on concepts that directly impact LLM inference performance. Graphics history, shader programming, and training-specific optimizations are omitted in favor of practical knowledge for serving systems engineers.

The GPU as a Parallel Compute Engine

The fundamental design difference between CPUs and GPUs is a trade-off between flexibility and parallelism. CPUs have few powerful cores optimized for low-latency sequential execution with complex control flow. GPUs have thousands of simpler cores optimized for executing the same operation on many data elements simultaneously.

An NVIDIA GPU contains multiple Streaming Multiprocessors (SMs), each responsible for executing groups of threads called warps. On modern architectures like A100 and H100, a warp consists of 32 threads that execute in lockstep: the same instruction runs on all 32 threads simultaneously, each operating on different data. This Single Instruction Multiple Data (SIMD) execution model is extremely efficient when all threads in a warp follow the same code path but suffers when threads diverge (for example, due to conditional branches where some threads take one branch and others take another).

Threads are organized hierarchically for programming purposes:

- **Thread**: The smallest unit of execution, running one instruction stream
- **Warp**: 32 threads executed simultaneously by one SM
- **Thread block**: A group of warps that can cooperate through shared memory and synchronization
- **Grid**: The complete set of thread blocks launched for a single kernel

When you launch a CUDA kernel to perform a matrix multiplication, you specify how many thread blocks to create and how many threads per block. The GPU runtime schedules these blocks across available SMs, launching warps as execution resources become available. A large matrix multiplication might launch thousands of blocks with millions of total threads, keeping all SMs busy for the duration of the computation.

This architecture explains why GPUs excel at certain workloads:

- **Data-parallel operations**: Matrix multiplications, element-wise operations, and convolution kernels map naturally to warp-level parallelism because the same computation applies uniformly across data elements.
- **Large problem sizes**: Thousands of threads keep SMs saturated, amortizing overhead and maximizing throughput.
- **Regular memory access patterns**: When threads in a warp access contiguous memory locations, the GPU can combine requests into efficient memory transactions.

And why GPUs struggle with others:

- **Control divergence**: Code with many conditional branches causes warps to serialize execution as different threads follow different paths.
- **Small problem sizes**: Launching a kernel with few threads underutilizes SMs and does not justify kernel launch overhead.
- **Irregular memory access**: Random memory accesses by different threads prevent coalescing and waste bandwidth.

LLM inference workloads are mostly well-suited to GPU execution because transformer operations (matrix multiplications, attention computation) are data-parallel and operate on large tensors. However, the decode phase presents challenges: processing a single token at each step means less parallelism than prefill, and KV cache access patterns can become irregular with variable-length sequences.

Memory Hierarchy: Registers, Shared Memory, Global Memory, HBM

GPU memory is organized in a hierarchy from fastest/smallest to slowest-/largest, similar to CPU caches but with different characteristics optimized for parallel access patterns.

Registers: Each thread has private registers for storing frequently accessed values during computation. Register access is essentially free (one cycle) but capacity is limited: each SM has a fixed number of registers shared among all active warps. Kernel design affects register usage: more complex kernels with many temporary variables use more registers per thread, which reduces the number of concurrent warps an SM can hold and potentially lowers occupancy.

Shared memory: Also called L1 cache in some contexts, shared memory is a small (typically 128 to 256 KB per SM on modern GPUs), fast memory region accessible by all threads in a block. Threads cooperate to load data from global memory into shared memory once and then access it repeatedly with low latency. This technique, called tiling, is fundamental to high-performance matrix multiplication kernels: the large input matrices are broken into tiles that fit in shared memory, reducing expensive global memory accesses.

Global memory: The main GPU memory accessible by all threads, backed by High Bandwidth Memory (HBM) chips stacked directly on the GPU package. Global memory has much higher latency than registers or shared memory (hundreds of cycles) but enormous capacity (40 to 80 GB on data center GPUs) and bandwidth (1.5 to 3.35 TB/s depending on GPU model).

HBM specifics: High Bandwidth Memory achieves its performance through wide memory interfaces: HBM2e on A100 uses 1024-bit buses, and HBM3 on H100 uses even wider interconnects. This contrasts with GDDR6 memory on consumer GPUs like the RTX 4090 or L40S, which uses narrower buses (384-bit) and achieves lower bandwidth despite similar or higher clock speeds.

The memory hierarchy directly impacts LLM inference performance through two mechanisms:

First, **arithmetic intensity** determines whether a computation is limited by compute capacity or memory bandwidth. Arithmetic intensity is the ratio of floating-point operations performed to bytes of data moved from global memory. Operations with high arithmetic intensity (large matrix multiplica-

tions) keep compute units busy while data transfers happen in the background, achieving performance close to peak compute throughput. Operations with low arithmetic intensity (small matrix-vector multiplications during decode) spend most time waiting for data and achieve performance limited by memory bandwidth.

Second, **data reuse** determines how effectively shared memory and caches reduce global memory traffic. Transformer attention during prefill reuses the same weight matrices across many tokens, allowing efficient caching. During decode, each step reads all model weights but processes only one new token, providing less opportunity for reuse and making the workload more sensitive to raw memory bandwidth.

Understanding this hierarchy explains why certain GPUs perform better for inference:

- H100's 3.35 TB/s HBM3 bandwidth versus A100's 2.04 TB/s HBM2e translates directly to faster decode performance because decode is memory-bound
- L40S's 864 GB/s GDDR6 bandwidth limits it significantly for large model inference despite having competitive compute capacity
- Consumer GPUs like RTX 4090 with 1 TB/s bandwidth perform surprisingly well for single-GPU inference but lack NVLink for multi-GPU scaling

CUDA Concepts Relevant to Inference

CUDA is NVIDIA's programming model for GPU computation. You do not need to write CUDA code to build inference systems, but understanding key concepts helps you understand what inference engines are doing under the hood and why certain configurations affect performance.

Kernels: A CUDA kernel is a function that runs on the GPU, launched from the CPU with a specified grid and block configuration. Each transformer layer in an LLM typically requires multiple kernel launches: one for the query-key-value projection, one for attention computation, one for the output projection, one for the feed-forward network, and so on. A complete forward pass through a 32-layer model might launch hundreds of individual kernels.

Kernel launch overhead is nontrivial: each launch requires the CPU to communicate with the GPU driver, validate arguments, and schedule execution. This overhead typically ranges from 5 to 10 microseconds per kernel launch. For decode steps where individual kernels do small amounts of work (processing a single token), launch overhead can consume a significant fraction of total time. Techniques like CUDA Graphs (discussed in Chapter 4) reduce this overhead by capturing and replaying kernel sequences without per-launch CPU intervention.

Streams: CUDA streams are queues of operations that execute asynchronously on the GPU. Multiple streams allow overlapping independent operations: while one stream executes a compute kernel, another can transfer data between CPU and GPU memory. For inference, streams enable pipelining where tokenization for the next request begins on the CPU while the current batch is being processed on the GPU.

Pinned memory: Also called page-locked memory, pinned memory is CPU memory that the operating system guarantees will not be paged to disk. This guarantee allows the GPU to use Direct Memory Access (DMA) for faster transfers across PCIe. Allocating pinned memory reduces CPU-GPU transfer latency but consumes system RAM that cannot be swapped and may impact CPU performance if overused. For inference systems where model weights reside on GPU and only small amounts of data move between host and device, pinned memory is typically used selectively for input buffers and output queues.

Tensor cores: Specialized hardware units in modern NVIDIA GPUs optimized for mixed-precision matrix multiplication operations. Tensor cores operate on small matrices (typically 4x4 or 8x8 tiles) at extremely high throughput, supporting FP16, BF16, FP8 and even lower precisions on newer architectures. Transformer models benefit enormously from tensor cores because attention and feed-forward computations are dominated by matrix multiplications that map directly to tensor core operations.

The H100's fourth-generation tensor cores with FP8 support deliver approximately 2,000 TFLOPS of FP8 throughput, roughly double the FP16 throughput of A100. This is why FP8 quantized inference on H100 can achieve significantly higher throughput than FP16 inference on the same hardware: the computation shifts from being limited by tensor core capacity to being limited by memory bandwidth for feeding those cores.

Why LLMs Are Memory-Bound (Mostly)

The observation that LLM inference is memory-bound requires careful qualification because different phases have different characteristics, but the decode phase where most generation time is spent is overwhelmingly limited by memory bandwidth rather than compute capacity.

Consider a single decode step for Llama 3 8B processing one token. The model has approximately 8 billion parameters, each stored in FP16 (2 bytes). Loading all weights from global memory requires reading roughly 16 GB of data. The arithmetic performed on this data: matrix multiplications and activations across 32 layers, produces perhaps 50 to 100 GFLOPS of computation for a single token.

The arithmetic intensity is therefore approximately:

50 GFLOPS / 16 GB = 3 FLOPs per byte

Compare this against the GPU's operational intensity ceiling (peak compute divided by peak bandwidth):

A100: 312 TFLOPS FP16 / 2.04 TB/s = 153 FLOPs per byte H100: ~1,979 TFLOPS FP8 / 3.35 TB/s = 590 FLOPs per byte

The decode workload's arithmetic intensity of 3 FLOPs/byte is far below either GPU's ceiling of 153 or 590 FLOPs/byte. The GPU has enormous unused compute capacity because it cannot get data to the compute units fast enough. This is the definition of memory-bound.

Prefill is different. Processing thousands of tokens simultaneously increases arithmetic intensity dramatically because the same weights are reused across all tokens in the batch. A prefill batch of 1,000 tokens performs roughly 1,000 times more computation while reading weights only once, pushing arithmetic intensity well above the memory-bound threshold into compute-bound territory. This is why prefill saturates tensor cores while decode leaves them largely idle.

The memory-bound nature of decode has several implications:

- **Memory bandwidth matters most**: When selecting GPUs for inference workloads where latency matters, prioritize memory bandwidth over raw TFLOPS. An H100's advantage over A100 comes primarily from its

68 percent higher memory bandwidth, not just its additional compute capacity.

- **Batching helps up to a point**: Increasing batch size during decode improves arithmetic intensity by reusing weight loads across more tokens, eventually pushing the workload closer to compute-bound and improving GPU utilization. However, larger batches also increase KV cache reads (which grow with total sequence length), creating a counterpressure that limits how far batching can help.
- **Quantization helps in two ways**: Reducing precision from FP16 to INT4 or FP8 both reduces the data volume that must be transferred (improving effective bandwidth) and increases arithmetic intensity by performing more operations per byte loaded.

The NVIDIA GPU Lineup for Inference

Selecting GPUs for inference requires balancing performance, memory capacity, interconnect capabilities, and cost. Here is a comparison of major NVIDIA data center GPUs relevant to LLM inference as of early 2026.

H100 SXM5 80GB: The current flagship for inference at scale. Key specifications:

- Memory: 80 GB HBM3 at 3,350 GB/s bandwidth
- Compute: ~2,000 TFLOPS FP8 with Transformer Engine; ~989 TFLOPS FP16
- Interconnect: NVLink 4 at 900 GB/s per GPU; supports NVSwitch for full mesh within HGX nodes
- Best for: Large models (30B to 70B parameters) where memory bandwidth limits decode performance; multi-GPU deployments requiring fast interconnect; FP8 inference workloads

A100 SXM4 80GB: The previous generation flagship, still widely deployed and cost-effective. Key specifications:

- Memory: 80 GB HBM2e at 2,039 GB/s bandwidth
- Compute: 312 TFLOPS FP16; no native FP8 tensor cores (FP8 via software emulation only)

- Interconnect: NVLink 3 at 600 GB/s per GPU; NVSwitch supported
- Best for: Models up to 30B parameters at FP16/BF16; cost-sensitive deployments where H100 pricing is prohibitive; workloads that do not require FP8 acceleration

L40S: A versatile GPU designed for graphics and AI inference hybrid workloads. Key specifications:

- Memory: 48 GB GDDR6 at 864 GB/s bandwidth
- Compute: 91 TFLOPS FP16; supports FP8 via Ada Lovelace tensor cores
- Interconnect: PCIe only, no NVLink
- Best for: Smaller models (7B to 14B parameters) where memory capacity is less critical; cost-sensitive deployments; environments requiring both graphics rendering and AI inference on same hardware

H200: An H100 variant with increased memory capacity. Key specifications:

- Memory: 141 GB HBM3e at 4,800 GB/s bandwidth
- Compute: Similar to H100
- Best for: Very large models or long-context workloads where 80 GB is insufficient; scenarios requiring maximum KV cache capacity per GPU

RTX 4090: A consumer GPU that punches above its weight for single-GPU inference. Key specifications:

- Memory: 24 GB GDDR6X at approximately 1,008 GB/s bandwidth
- Compute: ~1,480 TFLOPS FP8; strong tensor core performance
- Interconnect: PCIe only, no NVLink
- Best for: Development and prototyping; single-GPU deployments of small models (up to 13B quantized); cost-conscious setups where data center GPUs are unavailable

When comparing these GPUs for inference, memory bandwidth is often the more important specification than raw compute for the reasons explained above. The L40S has respectable compute numbers but its GDDR6 bandwidth limits it significantly compared to HBM-equipped GPUs for large model serving. The RTX 4090's strong performance relative to its price makes it attractive

for startups and individual developers, but lack of NVLink and lower memory capacity limit scaling options.

Cloud pricing as of early 2026 reflects these performance differences: H100 instances typically cost $1.50 to $3.90 per hour depending on provider and commitment level, A100 instances range from $1.00 to $2.50 per hour, and RTX 4090 instances can be found for $0.40 to $0.80 per hour. Bare metal deployments amortize hardware costs differently but follow similar relative pricing patterns.

Beyond NVIDIA: AMD, Intel, and Custom Silicon

NVIDIA dominates the LLM inference market through a combination of hardware performance, software maturity (CUDA ecosystem), and tooling quality (TensorRT-LLM, vLLM optimization). However, alternatives exist for cost-conscious deployments or organizations seeking to reduce vendor lock-in.

AMD MI300X: AMD's flagship AI accelerator with 192 GB HBM3 memory at approximately 5.3 TB/s bandwidth, exceeding H100 on both capacity and bandwidth. ROCm software stack provides CUDA-compatible programming model but lags in ecosystem maturity: fewer optimized kernels for specific models, less polished profiling tools, and smaller community support. As of early 2026, MI300X is production-viable for well-supported models (Llama family) but requires more engineering effort than NVIDIA equivalents.

Intel Gaudi 2 and Gaudi 3: Purpose-built AI accelerators with competitive performance on training workloads and growing inference support. Gaudi's software stack (Habana SDK) is less mature for LLM inference specifically, with limited support for advanced serving features like continuous batching and speculative decoding. Pricing can be attractive for organizations already invested in Intel infrastructure.

AWS Trainium/Inferentia: Custom silicon designed for AWS cloud workloads. Inferentia2 provides competitive price-performance for specific models optimized for the platform but requires committing to AWS infrastructure and accepting limited model support compared to NVIDIA's broad ecosystem. Best suited for AWS-native organizations with standardized model choices.

Google TPU: Primarily designed for training, TPUs have limited inference optimization for LLMs compared to GPUs. Google's internal serving infrastruc-

ture uses TPUs effectively but open-source tooling for TPU-based LLM serving is less mature.

The practical recommendation: NVIDIA remains the default choice for production LLM inference in 2026 due to ecosystem maturity, tool support, and predictable performance. Alternative platforms are worth evaluating when cost savings are substantial (30 percent or more), when you have engineering capacity to handle additional complexity, or when specific workload characteristics align with alternative hardware strengths (for example, MI300X's large memory for very long contexts).

Summary

GPU architecture fundamentally shapes LLM inference performance. GPUs excel at data-parallel matrix operations through thousands of threads executing in warps across Streaming Multiprocessors. Memory hierarchy from registers through shared memory to HBM determines whether workloads are compute-bound or memory-bound. LLM decode is predominantly memory-bound with arithmetic intensity far below GPU operational ceilings, making memory bandwidth the critical hardware characteristic for latency optimization. CUDA concepts like kernels, streams and tensor cores explain how inference engines map model computation to hardware. The NVIDIA GPU lineup offers different trade-offs between bandwidth, capacity, interconnect and cost that must be matched to workload requirements. Alternative platforms exist but currently require accepting ecosystem immaturity in exchange for cost or capacity advantages.

The next chapter examines the CPU-GPU interface: how data moves between host and device, why this boundary matters for latency, and techniques to minimize overhead at this critical junction.

Chapter 4: The CPU-GPU Interface

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Data Movement Patterns in Inference

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

PCIe Bandwidth and Latency

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Pinned Memory and Asynchronous Transfers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Zero-Copy and Unified Memory Considerations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Kernel Launch Overhead and CUDA Graphs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Chapter 5: Batching and Scheduling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Static Batching: Simple but Wasteful

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Continuous Batching (vLLM-style): The Breakthrough

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Dynamic Batching and Micro-batching

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Scheduling Policies: Fairness, Priority, and Latency

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Preemption and Speculative Execution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Implementing a Scheduler: Design Decisions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Chapter 6: Memory Management for LLMs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

KV Cache Size Calculations and Capacity Planning

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

PagedAttention and Block Management (vLLM)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Prefix Caching for Shared Contexts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

KV Cache Eviction Policies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Quantization: Compressing Weights and Activations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Memory Fragmentation and Defragmentation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Chapter 7: Advanced Decoding Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Sampling Methods and Their Latency Impact

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Speculative Decoding: Drafting Faster Than Verifying

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Medusa and Multi-Token Prediction Heads

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Early Exit and Layer Skipping

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Repetition Handling and Stopping Criteria

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Chapter 8: Parallelism Strategies for Large Models

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Tensor Parallelism: Splitting Matrices Across GPUs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Pipeline Parallelism: Splitting Layers Across GPUs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Data Parallelism for Inference: When and Why

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Sequence (Context) Parallelism: Splitting Long Prompts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Expert Parallelism for Mixture-of-Experts Models

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Hybrid Parallelism: Combining Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Chapter 9: Multi-GPU and Multi-Node Systems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

NVLink and NVSwitch: Intra-Node Communication

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

InfiniBand and RoCE: Inter-Node Communication

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

NCCL and Collective Communication

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Topology-Aware Placement and Scheduling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

RPC Frameworks for Distributed Serving

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Fault Tolerance in Distributed Inference

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Chapter 10: Building an Inference Server

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

From Single-Model Script to Production Server

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

API Design: OpenAI-Compatible Endpoints and Beyond

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Streaming Responses: Why and How

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Choosing an Inference Engine: vLLM vs SGLang vs TensorRT-LLM vs llama.cpp

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Containerization and Deployment Basics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Warmup, Preflight, and Readiness Probes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Chapter 11: Cluster-Scale Serving

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Load Balancing Strategies for Inference Traffic

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Request Routing: Model Selection and Workload Segmentation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Admission Control and Backpressure

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Autoscaling: Signals, Policies, and Pitfalls

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Multi-Model Serving on Shared Infrastructure

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Kubernetes and GPU Orchestration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Chapter 12: Disaggregated Architectures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Why Disaggregate Prefill and Decode?

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Architecture of a Disaggregated System

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Implementations and Research

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Trade-offs and Complexity

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Distributed Caching Layers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Chapter 13: Reliability and Observability

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Health Checking and Readiness for GPU Services

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Distributed Tracing for Inference Requests

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Metrics That Matter in Production

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Graceful Degradation Under Load

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Retry Logic, Timeouts, and Circuit Breakers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Incident Response and Debugging GPU Failures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Chapter 14: Capacity Planning and Cost Optimization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Workload Characterization: Understanding Your Traffic

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Capacity Calculations: From Tokens to GPUs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Cost Per Token and Cost Per Request Modeling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Right-Sizing Hardware for Your Workload

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Performance Regression Testing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Chapter 15: Extremely Low-Latency Serving

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Sub-100ms TTFT: Is It Achievable?

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Real-Time Conversational and Voice Workloads

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

CUDA Graphs and Kernel Fusion at Scale

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Communication/Computation Overlap in Distributed Systems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Edge and On-Device Inference for Ultra-Low Latency

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Chapter 16: Specialized Workloads and Emerging Techniques

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Long-Context Inference (100K+ Tokens)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Reasoning Models with Variable Output Lengths

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Multimodal Inference: Images, Audio, and Video

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

High-Concurrency Serving Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Heterogeneous GPU Fleets

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Inference-Aware Model Optimization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Chapter 17: End-to-End Reference Architectures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Architecture A: Single GPU Prototype (Proof of Concept)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Architecture B: Single Multi-GPU Server (Small Team Production)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Architecture C: Small Cluster (Startup/SMB Production)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Architecture D: Geographically Distributed Platform (Enterprise Scale)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Chapter 18: Designing Low-Latency LLM Systems from First Principles

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Requirements Gathering and Workload Characterization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Latency Budget Construction

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Architecture Selection Decision Tree

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Hardware Sizing and Procurement Strategy

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Implementation Sequencing: Build in Stages

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Optimization Priorities: Where to Spend Your Time

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Production Readiness Checklist

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Chapter 19: Conclusion: Enduring Principles for a Fast-Changing Field

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

The Core Mental Models

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Where LLM Infrastructure Is Heading

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

A Note on Engineering Philosophy

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

Closing Thoughts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.

References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildinglow-latencyllminfrastructure.