

A Practical Guide to **Inference** **Engineering**

Architectures, Optimization Strategies,
Performance, Reliability and Production Systems



RYAN WHITMORE

A Practical Guide to Inference Engineering

Architectures, Optimization Strategies, Performance,
Reliability and Production Systems

Steve Publications

This book is available at

<https://leanpub.com/apracticalguidetoinferenceengineering>

This version was published on 2026-10-03



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2026 Steve Publications

Contents

- Architectures, Optimization Strategies, Performance, Reliability and Production Systems 1**
- Introduction: The Inference Problem 2**
 - The Scale of Inference Today 2
 - Why Inference Is Hard 2
 - Inference vs Training: Fundamental Asymmetries 4
 - What This Book Covers (and What It Does Not) 5
 - How to Use This Book 6
- Chapter 1: Inference Workloads and Performance Fundamentals 9**
 - What Is an Inference Request: Structure, Inputs, Outputs, Metadata 9
 - Latency: Definitions, Components and Why Tail Latency Matters 10
 - Throughput and Utilization: The Tradeoff Landscape 12
 - Request Arrival Patterns: Poisson, Bursts, Seasonality and Shock Load 13
 - The Latency-Throughput-Cost Triangle 14
 - Performance Metrics That Matter: p50, p95, p99, TTFT, ITL, QPS, Tokens/sec 15
 - Capacity Planning Fundamentals: From Workload to Resource Estimates 17
- Chapter 2: Model Serving Architectures and Patterns 20**
 - The Serving Stack: API, Runtime, Model, Hardware 20
 - Monolithic Serving vs Microservice Serving 21
 - Online, Nearline and Batch Serving: When to Use Each 22
 - Stateful vs Stateless Serving Design 23
 - Request Routing and Load Balancing Strategies 24
 - Scaling Dimensions: Replication, Sharding, Partitioning 26
 - API Design for Inference: Synchronous, Asynchronous, Streaming, WebSockets 27

Chapter 3: Computing Hardware for Inference 30

 Compute Paradigms: Tensor Operations, Matrix Multiplication, Sparsity 30

 GPUs: Architecture, Memory Hierarchy, Streaming Multiprocessors,
 Tensor Cores 30

 TPUs and ASIC Accelerators: Fixed-function vs Configurable 30

 CPUs for Inference: When They Make Sense, Vector Instructions,
 Memory 30

 Memory Hierarchy and Bandwidth: DRAM, HBM, SRAM, PCIe 30

 Interconnects: NVLink, InfiniBand, Ethernet, NUMA Effects 31

 Hardware Cost Models: CapEx, OpEx, Power, Cooling 31

Chapter 4: Model Serving Systems and Runtimes 32

 What a Serving System Does: Beyond “Loading a Model” 32

 Request Lifecycle: From HTTP to Model Execution to Response 32

 Model Loading, Initialization and Warmup 32

 Inference Engines: TorchServe, Triton, vLLM, TensorRT Serving,
 ONNX Runtime and Others 32

 Runtime Responsibilities: Memory Management, Scheduling, Opti-
 mization 32

 Containerization and Deployment: Docker, Kubernetes Integration 33

 Choosing a Serving System: Evaluation Criteria 33

Chapter 5: Concurrency, Batching and Scheduling 34

 Why Batching Works: Amortizing Overhead and Exploiting Parallelism 34

 Static vs Dynamic Batching: Trade-offs and Implementation 34

 Continuous Batching: Mechanism, Benefits and Limitations 34

 Queueing Theory Basics for Inference: Arrival, Service, Waiting 34

 Scheduling Algorithms: FIFO, Priority, Fairness, ML-Specific Scheduling 34

 Concurrency Control and Thread Pool Design 35

 Admission Control and Load Shedding Strategies 35

 Overlapping Computation and Communication 35

Chapter 6: Memory Management and KV Cache Systems 36

 Memory in Inference: Model Weights, Activations, KV Cache, Tempo-
 raries 36

 The KV Cache: Purpose, Growth, Memory Implications 36

 Paged Attention: How It Works, Memory Savings, Implementation 36

 Memory Fragmentation and Compaction 36

 GPU Memory Management: Allocations, Caching, Pinning 36

Spilling KV Cache to Host Memory: Trade-offs	37
Memory-Aware Batching and Eviction Strategies	37
Chapter 7: Model Optimization: Quantization	38
Why Quantization Works: Redundancy in Learned Parameters	38
Fixed-Point Arithmetic: Formats, Ranges, Scaling Factors	38
Post-Training Quantization: PTQ Methods and Accuracy	38
Quantization-Aware Training: Mechanism and When to Use It	38
Per-Tensor, Per-Channel, Per-Token Quantization	38
Mixed Precision Strategies	39
Hardware Support and Compatibility	39
Quantization Impact: Accuracy, Latency, Memory, Power	39
Common Pitfalls and Debugging Quantization Issues	39
Chapter 8: Model Optimization: Pruning, Distillation and Compression 40	40
Pruning: Motivation, Mechanisms and Types (Unstructured, Structured, Layer)	40
Pruning Algorithms: Magnitude-Based, Gradient-Based, Iterative Methods	40
Knowledge Distillation: Teacher-Student Framework, Loss Functions	40
Pruning and Distillation Combined: Synergistic Benefits	40
Compression Formats and Packaging	40
Structured Sparsity and Hardware Acceleration	41
Accuracy Trade-offs and Calibration	41
When Optimization Is Worth the Effort	41
Chapter 9: Compilation and Graph Optimization	42
Computation Graphs and Their Optimization Opportunities	42
Graph Transformations: Constant Folding, Dead Code Elimination, Type Canonicalization	42
Operator Fusion: Mechanism, Patterns, Performance Impact	42
Memory Planning and Optimization: Allocation, Reuse, Graph Coloring	42
Kernel Selection and Specialization	42
Hardware-Aware Compilation: Loop Tiling, Blocking, Vectorization . .	43
End-to-End Compilers: TVM, MLIR, XLA, TensorRT	43
Compilation Time vs Inference Gain: The Trade-off	43
Chapter 10: Kernel-Level and Runtime Optimization	44
Understanding Compute Kernels: What They Do, How They Execute .	44
Memory Access Patterns: Coalescing, Bank Conflicts, Cache Locality	44

CONTENTS

Cache Optimization: Register, L1, L2, Shared Memory	44
Custom Kernel Development: When It Is Worth It	44
CUDA and GPU-Specific Optimization Techniques	44
CPU-Specific Optimizations: AVX, AMX, Loop Unrolling	45
Runtime Optimizations: JIT Compilation, Caching, Warmup	45
Profiling at the Kernel Level	45
Chapter 11: Performance Engineering Methodology	46
The Performance Engineering Loop: Measure, Profile, Hypothesize, Optimize, Validate	46
Establishing a Measurable Baseline	46
Profiling Tools and Techniques: System, Runtime, Model, Kernel Levels	46
Identifying Bottlenecks: Compute, Memory, Network, Queue	46
The Roofline Model for Inference	46
Benchmarking Correctly: Workload Representation, Warmup, Statis- tical Validity	47
Avoiding Benchmark Pitfalls and Misleading Measurements	47
Regression Detection and Continuous Performance Testing	47
Capacity Modeling and Saturation Analysis	47
Chapter 12: Distributed Inference and Parallelism Strategies	48
Why Distribute Inference: Memory, Throughput, Latency Requirements	48
Data Parallelism for Inference: Replicated Models, Load Distribution .	48
Tensor Parallelism: Mechanism, Communication Patterns, Use Cases	48
Pipeline Parallelism: Stages, Bubbles, Throughput Implications	48
Hybrid Parallelism: Combining Strategies	48
Distributed Communication: All-Reduce, All-Gather, Point-to-Point .	49
Fault Tolerance in Distributed Inference	49
Scaling Laws for Inference: Where Scaling Stops Helping	49
When Distributed Inference Is Overkill	49
Chapter 13: LLM Inference: Prefill, Decode and Beyond	50
Autoregressive Generation: Mechanics and Asymmetry	50
Prefill and Decode: Two Distinct Workloads	50
Attention Complexity: Quadratic Scaling and Its Consequences	50
Efficient Attention Mechanisms: FlashAttention, Linear Attention, Sparse Attention	50
Speculative Decoding: How It Works, Speedup Analysis, Error Handling	50
Long-Context Inference: Challenges, Strategies, Trade-offs	51

Multi-Turn Conversations and Session Management	51
LLM-Specific Metrics and Monitoring	51
Chapter 14: Advanced LLM Optimization Techniques	52
Prefix and Prompt Caching: Mechanism, Hit Rates, Memory Costs . .	52
Chunked Prefill: Balancing Latency and Throughput	52
Early Exit: Confidence-Based Premature Termination	52
Layer Skipping and Adaptive Computation	52
Request-Level Optimization: Priority, Slicing, Merging	52
Context Window Optimization: Summarization, Retrieval, Sliding Window	52
Tokenization Efficiency and Vocabulary Trade-offs	53
Multi-Model Systems: Routing, Cascading, Speculative Hierarchies . .	53
Chapter 15: Inference for Different Workload Classes	54
Computer Vision Inference: Image Models, Batching Patterns, Hardware Fit	54
Speech and Audio Inference: Streaming, Chunking, Real-Time Constraints	54
Recommendation and Ranking: Scale, Low Latency, Batch Scoring . .	54
Embedding Inference: High Throughput, Dimensionality, Caching . .	54
Multimodal Inference: Composed Workflows, Asynchronous Stages .	54
Conventional ML: Trees, Linear Models, XGBoost, LightGBM	55
Generalizable Principles vs Workload-Specific Techniques	55
Polyglot Inference Platforms	55
Chapter 16: Production Operations: Observability, SLOs and Reliability	56
Defining SLOs and SLIs for Inference: Latency, Availability, Quality . .	56
Metrics That Matter: What to Collect and Why	56
Logging for Inference: Request-Level Observability	56
Distributed Tracing for Inference Systems	56
Alerting: Signal vs Noise, Smart Alerting Policies	56
Health Checks and Self-Diagnostics	56
Fault Isolation and Blast Radius Control	57
Graceful Degradation Strategies	57
Incident Response for Inference Systems	57
Chapter 17: Deployment, Lifecycle Management and Versioning	58
Model Versioning and Identification	58
Deployment Strategies: Rolling, Blue-Green, Canary	58

CONTENTS

A/B Testing and Shadow Deployment for Models	58
Rollback and Rapid Recovery	58
Model Registry and Lifecycle Tracking	58
Configuration Management for Inference	58
Automated Testing: Functional, Performance, Drift	59
Security: Authentication, Authorization, Isolation, Data Protection . .	59
Abuse Prevention and Rate Limiting	59
Chapter 18: Cost and Infrastructure Efficiency	60
Cost Drivers in Inference: Hardware, Memory, Compute, Network, Energy	60
Cloud vs On-Prem vs Hybrid: Cost and Control Trade-offs	60
Instance Selection and Right-Sizing	60
Autoscaling Policies: Metrics, Thresholds, Hysteresis	60
Reserved, Spot and On-Demand Capacity Strategies	60
Utilization Optimization: Packing, Sharing, Overcommitting	61
Inference Cost Modeling: Per-Request, Per-Token, Per-User	61
Energy Efficiency and Environmental Considerations	61
Total Cost of Ownership: Beyond Infrastructure Bills	61
Quality-Cost Trade-offs: When to Spend More for Better Performance	61
Chapter 19: End-to-End System Design: Realistic Scenarios	62
Design Process: From Requirements to Architecture	62
Scenario 1: High-Traffic Chat API with Strict Latency SLOs	62
Scenario 2: Batch Embedding Pipeline for Search Indexing	62
Scenario 3: Real-Time Recommendation System at Scale	62
Scenario 4: Multi-Model Platform Serving Heterogeneous Workloads	62
Capacity Estimation Walkthrough: Step by Step	63
Hardware Selection Framework	63
Optimization Strategy Selection Decision Tree	63
Monitoring and Continuous Improvement	63
Common Architectural Anti-Patterns and How to Avoid Them	63
Conclusion: Principles, Patterns and the Future	64
Enduring Principles of Inference Engineering	64
Patterns That Recur Across Systems	64
Emerging Trends: New Hardware, New Models, New Techniques . . .	64
What to Learn Next	64
The Future of Inference: Speculation Grounded in Evidence	64

References 65

Architectures, Optimization Strategies, Performance, Reliability and Production Systems

About this book: Modern AI inference is no longer a side task tacked on to training pipelines. It is a distinct engineering discipline with its own performance constraints, reliability requirements, cost dynamics and failure modes. This book explains how production inference systems are designed, optimized, deployed and operated. It is written for software engineers, ML engineers, platform engineers and technical leads who need to build inference platforms that are fast, reliable, cost-efficient and understandable. You will learn not only what inference engineering techniques exist but why they work, how they work internally, when to use them, what trade-offs they introduce and how to measure their impact. The book progresses from first principles through sophisticated production-grade systems, always grounding guidance in quantitative reasoning, reproducible measurement and trade-off analysis rather than marketing claims or prescriptive recipes.

Introduction: The Inference Problem

The Scale of Inference Today

Inference is where machine learning meets the real world. It is the moment a trained model actually does something useful: recognizes a face in a photo, approves or flags a payment, recommends the next video to watch, or generates a response to a user question. Training builds the model. Inference is what the model does with its life.

The scale of modern inference is staggering and still accelerating. Hyperscale operators report billions of inference requests per day across their platforms. In 2025, inference workloads already consumed approximately 9 percent of global data center capacity, compared to about 14 percent for training. By 2030, projections suggest inference will grow to 37 percent of data center capacity, while training falls to 13 percent. This is not a marginal shift. It is a structural reallocation of compute demand, driven by the fact that training is a one-time expense per model while inference costs accumulate with every request, every user interaction, every prediction.

Consider a single high-traffic LLM API: if it serves 100,000 requests per day with an average response length of 300 tokens and an average prompt length of 200 tokens, it generates 50 million tokens daily. At scale, that is hundreds of thousands of GPU hours per month, running continuously. The economics are such that a 10 percent improvement in inference efficiency can save millions of dollars annually at a large organization.

This scale creates unique pressures. Systems must handle bursty traffic, unpredictable request sizes, varying user expectations and strict service level agreements, all while operating at hardware limits where every gigabyte of memory and every second of compute is accounted for. Inference is no longer a research problem. It is a systems engineering discipline.

Why Inference Is Hard

Inference is hard for reasons that do not apply to many other domains. The challenges are not just about speed. They are about a collision of conflicting requirements in a physically constrained environment.

First, inference systems must simultaneously optimize for latency, throughput and cost. These three goals pull in different directions. Reducing latency typically requires dedicating resources to individual requests, which lowers overall throughput and raises cost. Maximizing throughput usually means batching requests together, which adds wait time and increases tail latency. Minimizing cost often involves sharing hardware and using cheaper capacity, which reduces both latency and throughput under load. Any real system chooses a point on a multi-dimensional trade-off curve and the right choice depends on the workload, the business model and the user experience requirements.

Second, inference workloads are highly variable. Requests arrive irregularly. Some are tiny: a single image classification or a short classification label. Some are massive: a 100,000-token prompt for a complex reasoning task. Some requests finish in milliseconds; others take seconds or longer. This variability makes capacity planning and resource allocation difficult. Oversize and you waste expensive hardware. Undersize and you violate service level agreements.

Third, modern inference is fundamentally memory-bound much of the time, especially for large language models. During autoregressive text generation, the dominant cost is not performing calculations but loading model weights and attention cache from memory. This makes inference performance less sensitive to raw compute power than one might expect and more sensitive to memory bandwidth, memory capacity and how efficiently data moves through the system. Understanding and working with these physical constraints is central to inference engineering.

Fourth, inference systems must be reliable and observable in ways that go beyond traditional services. A database can fail fast and raise an error. A neural network can fail silently, returning a plausible-looking but incorrect prediction. Latency can degrade gradually as traffic increases, without any discrete failure event. Monitoring inference systems requires tracking not only availability and error rates but also tail latency distributions, model output quality, token-level performance metrics, resource utilization and cost per request. All of these

metrics must be correlated to understand what is happening when things go wrong.

Finally, inference infrastructure is expensive. GPUs and specialized accelerators cost tens of thousands of dollars each, consume significant power, generate heat that must be cooled and depreciate quickly. Idle hardware is a major cost in many organizations. Engineering good utilization without sacrificing user experience is a constant balancing act.

Inference vs Training: Fundamental Asymmetries

Training and inference are not just different phases of the same process. They are fundamentally different computational problems with different requirements, different constraints and different optimization priorities.

Training is an optimization problem. It repeatedly adjusts model parameters to minimize a loss function over a dataset. The key characteristic of training is that it has no strict latency requirement. A training job may take days, weeks, or months. That is acceptable as long as it eventually converges. Training is also typically a one-time or infrequent operation per model version. It benefits enormously from distributed parallelism, gradient accumulation and fault tolerance.

Inference is a serving problem. It must return a prediction for new data quickly, reliably and at scale. The key characteristic of inference is that it is latency-sensitive. Users, applications, or downstream systems are waiting for results. Inference must also be continuous. It runs forever, every day, every hour, every minute. The cost model is different: instead of a one-time compute bill, inference generates ongoing operational expenses proportional to request volume.

These asymmetries have practical consequences:

- Training is compute-intensive and memory-intensive in ways that are relatively predictable given the batch size and model architecture. Inference faces variable workloads where batch sizes and sequence lengths change dynamically, making resource planning harder.
- Training can afford to use full precision and can benefit from large batches. Inference must often use reduced precision and small or variable batches to meet latency requirements.

- Training benefits from gradient computation and backward passes. Inference is purely a forward pass, which eliminates some overhead but also limits some optimization opportunities.
- Training is typically done by a small team of ML engineers. Inference is consumed by everyone: application developers, product teams, external users. The operational requirements for inference are correspondingly higher.

Many of the mistakes in inference engineering come from treating inference as simply training without the backward pass. It is not. The entire optimization landscape shifts.

What This Book Covers (and What It Does Not)

This book is about inference engineering as a complete discipline. It covers the full lifecycle from workload characterization to production operation, including:

- The foundational concepts needed to reason about inference: workload types, traffic patterns, latency and throughput, tail latency, capacity planning and performance metrics.
- System architectures and serving patterns: monolithic and distributed serving, online versus batch inference, request routing, load balancing and scaling strategies.
- Hardware for inference: GPUs, CPUs, TPUs, specialized accelerators, memory hierarchies, interconnects and how their physical properties shape system design.
- Model serving systems and runtimes: how serving frameworks work internally, their roles in the stack and how to evaluate and choose them.
- Concurrency, batching and scheduling: the techniques that allow inference systems to achieve high throughput and utilization, including dynamic batching, continuous batching, queueing theory and admission control.
- Memory management: how inference uses memory, the key-value cache in language models, paged attention and memory-efficient strategies.
- Model optimization: quantization, pruning, distillation and compression, with detailed explanations of mechanisms, accuracy trade-offs and practical considerations.

- Compilation and graph optimization: how compilers transform computation graphs, fuse operations, optimize memory and generate efficient code for specific hardware.
- Kernel-level optimization: low-level techniques for maximizing hardware performance, including memory access patterns, cache utilization and custom kernels.
- Performance engineering methodology: a systematic process for measuring, profiling, identifying bottlenecks, optimizing and validating improvements.
- Distributed inference: strategies for running inference across multiple GPUs and nodes, including tensor parallelism, pipeline parallelism and data parallelism.
- LLM-specific inference: the unique challenges of large language models, including prefill-decode asymmetry, efficient attention mechanisms, speculative decoding and prefix caching.
- Workload-specific considerations: how inference requirements differ across computer vision, speech, recommendation systems, embeddings, multimodal models and conventional ML.
- Production operations: observability, SLOs, monitoring, deployment strategies, versioning, rollback, security and reliability engineering.
- Cost and infrastructure efficiency: modeling inference costs, optimizing spend, capacity planning and the relationship between performance, quality and cost.
- End-to-end system design: realistic scenarios showing how to move from requirements to a production-ready inference system.

The book does not cover training methodology in depth. It assumes basic familiarity with machine learning concepts such as neural networks, gradient descent and model evaluation. It is not a tutorial on how to train models or a comparison of model architectures. It focuses exclusively on the engineering challenges of serving models in production.

The book is vendor-neutral in its framing. It discusses specific frameworks, runtimes and hardware where relevant, but always with the goal of explaining architectural concepts rather than endorsing specific products. The principles in this book apply regardless of whether you run on NVIDIA, AMD, Intel, or custom accelerators and regardless of which serving framework you choose.

How to Use This Book

This book is designed to be read sequentially, as each chapter builds on concepts introduced in earlier chapters. The first few chapters establish the foundational vocabulary and concepts needed throughout. The middle chapters explore specific techniques, architectures and optimization strategies in depth. The final chapters bring everything together in the context of production operations and end-to-end system design.

If you already have experience with ML inference and want to use this book as a reference, you can skip around to the chapters most relevant to your immediate problem. A few quick suggestions:

- If you are building an inference system from scratch, start from the beginning and read through at least Chapter 5 to understand the full landscape before making architecture decisions.
- If you are experiencing latency or throughput problems, Chapters 5, 11 and 13 provide detailed guidance on diagnosing and resolving performance issues.
- If you are trying to reduce inference cost, Chapters 13, 17 and 18 cover optimization techniques and cost engineering in detail.
- If you are working with large language models specifically, Chapters 13 and 14 are essential.
- If you are responsible for production reliability, Chapters 16 and 17 cover the operational practices you need.

Throughout the book, technical claims are grounded in evidence: peer-reviewed papers, official documentation, engineering publications and empirical measurements. Where evidence is limited or techniques are still experimental, this is stated explicitly. The book avoids marketing language and hype, focusing instead on mechanisms, trade-offs and practical engineering judgment.

The book contains no exercises or quizzes. Instead, each chapter aims to provide enough depth and context that you can apply the concepts directly to your own systems and make informed decisions based on your specific requirements and constraints.

The discipline of inference engineering is maturing rapidly. New techniques, hardware and frameworks appear frequently. The goal of this book

is not to catalog every current technology but to explain the underlying principles that will remain relevant as the field evolves. If you understand those principles, you will be equipped to evaluate new technologies critically and integrate them into your systems when they genuinely offer improvements.

Chapter 1: Inference Workloads and Performance Fundamentals

What Is an Inference Request: Structure, Inputs, Outputs, Metadata

Every inference system exists to answer a question: what happens when a request arrives? The answer depends on understanding the request itself in engineering terms, not just in machine learning terms. An inference request is not simply input data passed to a model. It is a unit of work with structure, constraints and context.

At its core, an inference request contains three elements:

1. **Input data:** the primary payload the model consumes. This might be an image, a text prompt, an audio stream, a feature vector, or a structured document. The size, format and preprocessing requirements of the input strongly affect inference behavior.
2. **Model parameters:** the algorithmic configuration that determines how the model processes the input. For a classifier, this might include the target class set and probability threshold. For a language model, this includes temperature, top-k or top-p sampling parameters, maximum token length, repetition penalties and system prompts. These parameters do not change the model weights but change how the model uses them.
3. **Metadata:** contextual information about the request that the serving system needs but the model does not consume. This includes request identifiers, user or tenant identifiers, priority levels, trace IDs, source service information and expected response format. Metadata is essential for routing, load balancing, observability, billing and security, but adds processing overhead.

The structure of an inference request determines much of what follows. A short classification request with fixed-size inputs can be served very differently from a long-context language model request with variable-length output.

The first might complete in microseconds with a batch of thousands processed together. The second might take seconds per request, cannot be easily batched with other requests and grows in cost as the output length increases.

Requests also carry implicit constraints that may not be explicitly stated in the API. A real-time fraud detection model has a hard latency budget: if the prediction takes longer than 50 milliseconds, the system must fall back to a default action. A chat application has a softer but still meaningful constraint: time to first response should be under a few seconds and token generation should be smooth enough to feel natural. A batch image processing job may have no per-request latency requirement but a deadline for completing a million images overnight.

Understanding the request at this level of detail is the foundation for everything else: architecture, capacity planning, optimization, monitoring and incident response. You cannot design a system for workloads you have not fully characterized.

Latency: Definitions, Components and Why Tail Latency Matters

Latency is the most discussed and most misunderstood metric in inference engineering. The common definition is simple: the time from when a request arrives until the response is complete. The engineering reality is much more complex because latency is not a single thing. It is composed of multiple phases, each with different drivers and each phase matters differently depending on the workload.

A complete latency breakdown for an inference request typically includes:

- **Network latency:** the time for the request to travel from client to server and the response to travel back. This depends on physical distance, network topology, protocol overhead and congestion. In many systems, this is a small component but can dominate in geographically distributed deployments.
- **Queue time:** the time the request spends waiting in a queue before computation begins. Queue time is entirely determined by system load and scheduling policy. Under light load it is negligible; under heavy load it can dominate all other components. Queue time is also where tail

latency problems are most visible: as utilization approaches capacity, queue lengths grow non-linearly and a small fraction of requests wait much longer than the rest.

- **Preprocessing time:** the time to validate, parse and transform the input into the format the model expects. This might include tokenization for text, image resizing and normalization, audio feature extraction, or feature engineering for tabular data. Preprocessing is often CPU-bound and can be a bottleneck if not designed carefully, especially for large inputs.
- **Model execution time:** the time the model spends computing the prediction. This is the core inference cost and includes all tensor operations, memory transfers and control flow within the model. It is typically split further into compute time (actual arithmetic operations), memory access time (loading weights and activations) and communication time (in distributed systems).
- **Postprocessing time:** the time to format, filter and package the output for the client. This might include decoding logits into text, applying business logic filters, formatting JSON responses, or translating model outputs into domain-specific results.
- **Serialization and transmission time:** the time to serialize the response and send it over the network. For large outputs (such as long text generation or high-resolution images), this can be significant.

Understanding latency composition is important because optimization efforts should target the dominant component. If queue time accounts for 80 percent of end-to-end latency under peak load, then optimizing the model kernel will not help users who are waiting in the queue. If preprocessing dominates, then model-level optimizations are misallocated effort.

The most critical aspect of latency in production systems is not average latency but tail latency. Tail latency refers to the high-percentile response times: the times experienced by the slowest fraction of requests, typically measured at p95, p99, or p99.9.

Tail latency matters for several reasons. First, users do not experience averages. They experience individual requests. If 99 percent of requests complete in under 200 milliseconds but the slowest 1 percent take 5 seconds, the user experience is defined by those 5-second waits, especially because they tend to occur at inconvenient moments. Second, in systems composed of multiple services, tail latency compounds. If each of ten microservices in a

request path has a p99 of 200 milliseconds, the end-to-end p99 is not 2 seconds but potentially much worse, because the tail of each service contributes independently. Third, tail latency is often a symptom of deeper problems: resource contention, garbage collection pauses, cache misses, scheduling delays, or hardware issues. Average latency can look fine while tail latency reveals underlying instability.

In inference systems, tail latency is particularly dangerous because the workload is variable. A request with an unusually long input, an unusually complex generation task, or a rare edge case can take orders of magnitude longer than a typical request. Without careful design, such requests can block resources and degrade performance for everyone.

Throughput and Utilization: The Tradeoff Landscape

Throughput and utilization are two sides of the same coin. Throughput is the rate at which the system completes requests: requests per second, tokens per second, or images per second, depending on the workload. Utilization is the fraction of available capacity that is actively doing useful work: the percentage of time GPUs are computing, CPUs are processing, or network links are transmitting.

The relationship between throughput and utilization is governed by basic queuing principles. At low utilization, throughput scales roughly linearly: double the traffic, double the throughput, with minimal impact on latency. At high utilization, the system enters saturation: throughput grows more slowly than traffic and latency increases dramatically. The exact shape of this curve depends on the system architecture, scheduling policy and workload characteristics, but the fundamental behavior is universal.

There is no such thing as free throughput. Every gain in throughput requires either more resources or longer latency (or both). This is the latency-throughput trade-off that dominates inference engineering. The trade-off is not a dial to tune. It is a constraint curve defined by the physics of the system. You cannot have both minimal latency and maximum throughput simultaneously.

Batching is the primary mechanism for moving along this curve. By grouping multiple requests into a single computation, batching increases the fraction of time the hardware spends doing useful compute rather than idle overhead,

thereby increasing both throughput and utilization at the cost of adding wait time for requests that must wait for a batch to form. Static batching uses fixed batch sizes and waits until the batch is full or a timeout expires. Dynamic batching adjusts batch sizes based on current conditions. Continuous batching, used in advanced LLM serving systems, adds and removes requests from batches on a per-step basis, achieving high utilization without requiring all requests to have the same duration.

The choice of where to operate on the latency-throughput curve is a business decision as much as an engineering decision. A real-time fraud detection system must prioritize low latency and accept lower utilization and higher cost. A batch recommendation generation job can prioritize throughput and utilization and tolerate longer completion times. A chat API must balance both: low enough latency to feel responsive, high enough throughput to handle many concurrent users.

Request Arrival Patterns: Poisson, Bursts, Seasonality and Shock Load

Inference systems do not face uniform, predictable traffic. Request arrival patterns are one of the defining characteristics of real-world workloads and they have enormous implications for system design, capacity planning and reliability.

Four categories of arrival patterns are most relevant:

Poisson arrivals represent the simplest case: requests arrive independently at a constant average rate. This is a reasonable approximation for many background workloads but rarely holds exactly in production. Poisson arrivals are analytically tractable and form the basis for standard queuing models (M/M/c, M/G/c). If your workload approximates Poisson, you can use well-known formulas to estimate queue lengths, waiting times and required capacity.

Bursty arrivals are far more common. Traffic comes in surges separated by quieter periods. This happens because user behavior is not uniform. People interact with systems at particular times: when a product launches, when a marketing campaign goes live, when a breaking event occurs, when a batch job triggers a cascade of requests. Bursts challenge systems because capacity is typically sized for average or even p95 load, not for peak load. When a burst

arrives, queues build rapidly, latency spikes and the system may fail entirely if protection mechanisms are inadequate.

Seasonal patterns operate on predictable cycles: daily, weekly, monthly. E-commerce traffic peaks during holidays. Gaming traffic peaks during weekends. Professional services see daytime peaks in business hours. Seasonality is manageable through planning: you can provision extra capacity ahead of known peaks, use scheduled autoscaling, or shift non-urgent workloads to off-peak hours. The challenge is accurately predicting the magnitude of seasonal changes, which can shift due to external factors.

Shock loads are sudden, large-scale increases in traffic that are neither predicted nor planned for. A viral social media post, a news event, a system failure causing retries, or a denial-of-service attack can generate shock loads that exceed designed capacity by orders of magnitude. Systems cannot be sized for arbitrary shocks without enormous waste, so they must handle them gracefully through mechanisms like load shedding, rate limiting, circuit breakers and degradation modes. A well-designed system does not try to survive a shock load unchanged. It protects itself and degrades in a controlled manner.

Understanding your arrival pattern is essential for capacity planning and architectural choices. A system that receives steady Poisson-like traffic can operate at high utilization with minimal headroom. A system subject to frequent bursts must maintain lower baseline utilization to absorb peaks. A system vulnerable to shock loads must implement robust protection mechanisms and define clear degradation policies.

The Latency-Throughput-Cost Triangle

Every inference system faces a triangle of competing objectives: latency, throughput and cost. You can optimize for two of these at the expense of the third, but not all three simultaneously. This is a fundamental constraint, not a temporary limitation.

Consider the three edges of the triangle:

- **Low latency and high throughput** requires massive hardware overprovisioning. Every request gets dedicated resources, batching is minimal, queue times are negligible and the system never nears saturation. This

is extremely expensive and only feasible for the most latency-critical, highest-value workloads.

- **Low latency and low cost** requires sacrificing throughput. The system has limited resources. When traffic exceeds capacity, requests must be rejected or queued for longer than acceptable, reducing effective throughput. This is the model for niche, high-margin services that accept limited scale.
- **High throughput and low cost** requires sacrificing latency. The system maximizes resource utilization through aggressive batching, shared infrastructure and efficient scheduling. Queue times are longer and tail latency is higher. This is the model for batch processing, nearline analytics and workloads where latency is not a primary concern.

The practical art of inference engineering is positioning a system at the right point on this triangle for its specific requirements, then continuously adjusting that position as requirements and workloads evolve. There is no universally correct position. A fraud detection model should be on the low-latency edge. A nightly batch embedding generation job should be on the low-cost edge. A consumer-facing chat API should be somewhere in the middle, biased toward latency but with significant throughput requirements.

A common mistake is to try to optimize for all three objectives simultaneously without explicit trade-offs. This results in systems that are expensive, complex and mediocre at everything. Explicitly stating which objectives take priority simplifies design decisions and leads to better systems.

Performance Metrics That Matter: p50, p95, p99, TTFT, ITL, QPS, Tokens/sec

A production inference system must be measured and it must be measured with the right metrics. Different workloads require different metrics, but several core measures are universally important.

Latency percentiles (p50, p95, p99) are the most fundamental metrics. The p50 (median) represents the typical request experience. The p95 represents the experience of the slowest 5 percent of requests. The p99 represents the slowest 1 percent. These percentiles reveal the distribution of response times and expose problems that averages hide. A system with p50 of 100ms and p99

of 2 seconds has a different problem than a system with p50 of 100ms and p99 of 300ms. The first has occasional catastrophic delays; the second is uniformly responsive.

For LLM inference specifically, latency decomposes into more detailed metrics:

- **Time to first token (TTFT):** the time from request submission until the first output token is returned. This is the latency users perceive as thinking time. TTFT is dominated by the prefill phase (processing the input prompt), queue time and network latency. It is the most important metric for user-perceived responsiveness.
- **Inter-token latency (ITL):** the time between consecutive output tokens during generation. This determines how smoothly the output appears. ITL is dominated by the decode phase and memory bandwidth. Consistent ITL creates a natural streaming feel; variable ITL creates stuttering.
- **Time per output token (TPOT):** the average time to generate one token, computed as total generation time divided by number of tokens. TPOT and ITL are related but distinct: TPOT is a per-request average, while ITL measures individual intervals.
- **End-to-end latency:** the total time from request submission to complete response. This is TTFT plus the time to generate all remaining tokens.

Throughput metrics measure system capacity:

- **Requests per second (RPS or QPS):** the number of requests completed per second. This is the standard metric for most workloads.
- **Tokens per second (TPS):** the total number of output tokens generated per second across all requests. For LLMs, this is often more informative than RPS because requests have variable output lengths.
- **System TPS vs user TPS:** system TPS measures total capacity, while user TPS (tokens per second per user) measures individual responsiveness. As concurrency increases, system TPS typically rises while user TPS falls, because resources are shared.

Utilization metrics reveal how efficiently resources are used:

- **GPU/CPU utilization:** the fraction of time accelerators or processors spend on useful compute. High utilization is generally good for cost but can indicate the system is near saturation.

- **Memory utilization:** the fraction of available memory in use. Critical for large models where memory capacity limits batch size and concurrency.
- **Network utilization:** the fraction of network bandwidth consumed. Important in distributed systems where inter-node communication can be a bottleneck.

Quality metrics ensure the system is correct:

- **Error rate:** the fraction of requests that fail. Includes timeouts, internal errors and rejected requests.
- **Goodput:** the fraction of requests that meet defined quality criteria, such as latency thresholds or accuracy requirements. Goodput is often more meaningful than raw throughput because it accounts for requests that completed but performed poorly.

Cost metrics track economic efficiency:

- **Cost per request:** infrastructure cost divided by number of requests.
- **Cost per token:** for LLMs, cost divided by tokens generated.
- **Cost per unit of business value:** the most meaningful cost metric, but also the hardest to define. Examples include cost per classified document, cost per recommended item, or cost per completed conversation.

No single metric tells the whole story. A system optimized for one metric can degrade others. The best monitoring systems track multiple metrics and correlate them to understand system behavior holistically.

Capacity Planning Fundamentals: From Workload to Resource Estimates

Capacity planning is the process of determining how much hardware and infrastructure is needed to handle expected traffic while meeting performance requirements. Good capacity planning is quantitative, evidence-based and iterative. It does not start with guesses. It starts with measurements.

The basic capacity planning workflow is:

1. **Characterize the workload:** measure the expected request rate, input size distribution, output size distribution and latency requirements. If you are building a new system, use historical data from similar workloads, load tests, or prototypes. Never rely solely on anecdotal estimates.
2. **Measure per-request cost:** run a controlled experiment with the model on representative hardware to determine how long a typical request takes, how much memory it uses and how much compute it consumes. Repeat for edge cases: the largest requests, the smallest requests and the median request.
3. **Calculate baseline capacity:** divide expected request volume by per-request cost. For example, if you expect 1,000 requests per second and each request takes 100 milliseconds of GPU time on average, you need 100 GPU-seconds per second, or 100 fully utilized GPUs.
4. **Account for utilization targets:** systems cannot operate at 100 percent utilization without unacceptable latency. A typical target utilization for latency-sensitive systems is 50 to 70 percent. Adjust capacity accordingly. If baseline is 100 GPUs at 100 percent utilization and you target 60 percent utilization, you need approximately 167 GPUs.
5. **Add headroom for variability:** account for traffic spikes, burst patterns and growth. A common practice is to add 20 to 50 percent headroom depending on workload volatility and tolerance for overload.
6. **Validate with load testing:** deploy the system and run controlled load tests that simulate expected traffic patterns. Measure actual latency, throughput and utilization. Compare with estimates and adjust capacity.
7. **Iterate:** capacity planning is not a one-time activity. As workloads change, as models change and as optimization efforts improve efficiency, capacity requirements change. Re-evaluate regularly.

A more sophisticated capacity planning approach uses queuing theory to model the relationship between arrival rate, service time, number of servers and expected latency. For a system with Poisson arrivals and exponentially distributed service times (M/M/c queue), the expected waiting time in the queue is:

$$W_q = (\rho^c / (c * (c - \rho))) * (1 / \mu)$$

where c is the number of servers, λ is the arrival rate, μ is the service rate per server and $\rho = \lambda / (c * \mu)$ is the utilization. This formula shows that as utilization approaches 1, waiting time grows non-linearly. It also

shows that adding servers reduces waiting time dramatically when the system is heavily loaded.

Queuing models are approximations and real workloads rarely follow their assumptions perfectly. Nevertheless, they provide valuable intuition: the cost of operating near capacity is disproportionately high in terms of latency. This is why systems must maintain headroom and why capacity planning must account for variability, not just averages.

Good capacity planning is also specific to the workload type. An LLM serving system must plan for variable-length outputs and the memory costs of key-value caches. A computer vision system must plan for image sizes and batch sizes. A recommendation system must plan for feature fetches and model cascades. The core principles are the same, but the details differ.

Chapter 2: Model Serving Architectures and Patterns

The Serving Stack: API, Runtime, Model, Hardware

Every inference system is built in layers. Understanding the serving stack clarifies responsibilities, failure boundaries and optimization opportunities. At a minimum, there are four layers: the API layer, the runtime layer, the model layer and the hardware layer.

The API layer is the interface through which clients interact with the inference system. It handles HTTP or gRPC requests, authentication, authorization, request validation, rate limiting, response formatting and error handling. The API layer is responsible for translating between client expectations and internal representations. It is often implemented as a lightweight web service that delegates inference to the runtime layer. The API layer is where you implement idempotency, retries, timeouts and protocol-specific behaviors such as streaming responses.

The runtime layer is where inference actually happens. It loads the model into memory, manages the computation graph, performs batching, handles concurrency and executes the model on the underlying hardware. The runtime layer is responsible for performance optimization, memory management and hardware utilization. Frameworks such as NVIDIA Triton Inference Server, vLLM, TensorRT-LLM, ONNX Runtime and custom serving implementations all provide runtime capabilities. The runtime layer is where most inference engineering work occurs, because it is where architectural decisions about batching, memory and scheduling directly impact performance.

The model layer is the machine learning artifact itself: the weights, architecture definition, preprocessing configuration and postprocessing logic. The model is trained by data scientists and ML engineers, then consumed by the serving system. A well-engineered serving system treats models as deployable artifacts with versioning, metadata and lifecycle management. The model layer is typically decoupled from the API and runtime layers: the same model can be

served by different runtimes and different models can be served by the same runtime.

The hardware layer provides the physical compute, memory, storage and networking resources on which inference executes. This includes GPUs, CPUs, TPUs, specialized accelerators, NVMe storage, network interfaces and interconnects. The hardware layer constrains everything above it: memory capacity limits model size and batch size, memory bandwidth limits decode speed, compute capacity limits throughput and network bandwidth limits distributed system efficiency. Good architecture design works with hardware constraints, not against them.

These layers are not rigid. In simple deployments, a single Python process might serve as API, runtime and model host simultaneously. In complex deployments, each layer is distributed across many processes and machines, with dedicated services for load balancing, model management, monitoring and configuration. The degree of separation between layers determines the system's complexity, flexibility and operational overhead.

Monolithic Serving vs Microservice Serving

One of the first architectural decisions when building an inference system is whether to deploy it as a monolith or as a collection of microservices.

A monolithic serving architecture packages the API, runtime, model loading and supporting logic into a single deployable unit. When a client makes a request, it goes to one service that handles everything. This approach is simple, fast and easy to reason about. There is no inter-service communication overhead. Deployment is straightforward: update the monolith, redeploy, done. Monitoring is centralized. Debugging is easier because all the code is in one place.

A microservice serving architecture decomposes the system into multiple specialized services. The API gateway handles protocol, authentication and routing. Separate services handle different models or model families. Preprocessing might be its own service. Postprocessing, caching and logging might be independent services. Each service can be deployed, scaled and updated independently.

The choice between monolith and microservices is not about which is better in principle. It is about which fits your requirements.

Monolithic serving is appropriate when:

- You are serving a single model or a tightly coupled set of models.
- Latency is critical and inter-service communication overhead matters.
- You have a small or medium-scale deployment where operational complexity is a concern.
- The model and its serving requirements are stable and not changing rapidly.

Microservice serving is appropriate when:

- You are serving many independent models with different scaling, update and latency requirements.
- Different teams own different models and need independent deployment control.
- Some components need to scale independently: for example, a pre-processing service handling image transformations might need different capacity than a classification model.
- You require fine-grained fault isolation: a failure in one model should not bring down all models.
- You need language or framework diversity: different models may be best served by different runtimes.

The cost of microservice architecture is complexity. Every added service introduces network latency, potential failures, deployment coordination, monitoring overhead and debugging challenges. For small-scale inference workloads, microservices often add more problems than they solve. For large, heterogeneous model portfolios, they provide necessary flexibility.

A pragmatic middle ground is the modular monolith: a single deployable unit with clear internal boundaries and interfaces, designed so that components can be extracted into separate services later if needed. This approach avoids premature distributed complexity while preserving the option to evolve.

Online, Nearline and Batch Serving: When to Use Each

Inference workloads fall into three broad latency categories: online, nearline and batch. Each category has different architectural requirements, optimization priorities and cost characteristics.

Online serving handles requests that must be answered immediately, typically within milliseconds to a few seconds. The client is waiting. Examples include fraud detection during payment processing, real-time search ranking, conversational AI chat and live recommendation personalization. Online serving systems must prioritize low latency and high availability. They run on fast hardware (often GPUs or specialized accelerators), use minimal batching and are sized to handle peak load. Cost per request is high because capacity must be reserved for burst traffic even when utilization is low.

Nearline serving handles requests that must be answered relatively quickly but can tolerate some delay, typically from a few seconds to a few minutes. Examples include email spam classification after delivery, document processing after upload and recommendation pre-computation triggered by user activity. Nearline serving systems can use more aggressive batching, share resources with other workloads and tolerate occasional delays. Cost per request is lower than online serving because systems can operate at higher utilization and use cheaper hardware.

Batch serving processes large volumes of data at scheduled times with no strict per-request latency requirement. Examples include nightly scoring of millions of customer profiles, embedding generation for a search index, offline model evaluation and data labeling for training. Batch serving systems prioritize throughput and cost efficiency over latency. They can use maximum batch sizes, schedule around other workloads and run on spot or reserved capacity. Cost per request is the lowest of the three categories.

These categories are not mutually exclusive. A single system might support multiple modes: an online endpoint for real-time predictions with a batch job that pre-computes results for the most common inputs. A recommendation system might have an online ranking model that scores items in milliseconds, backed by a nearline retrieval model and a batch embedding generator.

The key insight is that mixing modes without proper isolation is dangerous. A batch job that exhausts GPU memory can starve online requests. An online traffic spike can delay batch processing. Systems that support multiple modes should isolate resources, implement priority scheduling and define clear policies for what happens when capacity is contended.

Stateful vs Stateless Serving Design

Most machine learning models are mathematically stateless: given the same input, they produce the same output. However, serving systems often introduce state, either explicitly or implicitly. Understanding the distinction between stateful and stateless serving is essential for designing systems that scale and recover from failures correctly.

Stateless serving means each request is processed independently, using only the information contained in the request itself. The system holds no memory of previous requests beyond the loaded model weights. Stateless serving is simpler to design, easier to scale and more resilient to failures. Any replica can handle any request. If a replica fails, another takes over without data loss. Stateless systems are the default choice for classification, regression, computer vision and most conventional inference workloads.

Stateful serving means the system maintains information across requests. This is necessary for certain workloads. Autoregressive language models, for example, maintain a key-value cache of previous tokens during generation, which is state relative to a single request but is cleared when the request completes. Multi-turn conversational systems may need to store conversation history across requests. Session-based recommendation systems track user interactions over time. Recurrent neural networks used for sequence processing carry hidden state from one timestep to the next.

State introduces complexity. Stateful systems must handle consistency: if a request is rerouted to a different replica, the state must be available. They must handle failure: if a replica holding state crashes, that state must be recoverable or the request must be retried from a known checkpoint. They must handle scalability: state grows with usage and eventually must be evicted or partitioned.

The general principle is to minimize state wherever possible. When state is unavoidable, design it explicitly with clear semantics: which requests share state, how long state persists, what happens on eviction and how state is recovered after failure. Stateless architectures are easier to operate. Stateful architectures are harder but sometimes necessary.

Request Routing and Load Balancing Strategies

When an inference system runs multiple replicas, requests must be distributed among them. This is the job of request routing and load balancing. The strategy used directly affects latency, throughput, fairness and resource utilization.

Simple load balancing strategies include:

- **Round-robin:** requests are distributed cyclically across replicas. Simple and fair, but does not account for replica load or request cost. Works well when all replicas are similar and all requests have similar cost.
- **Random:** each request is sent to a randomly selected replica. Similar to round-robin in behavior, with slightly different variance characteristics. Useful when simplicity is prioritized.
- **Least connections:** the request is sent to the replica with the fewest active connections. More adaptive than round-robin because it accounts for current load. Works well when requests have variable duration.
- **Weighted round-robin:** replicas are assigned weights proportional to their capacity and requests are distributed accordingly. Useful when replicas have different hardware or configurations.

Advanced strategies are designed for the specific characteristics of inference workloads:

- **Latency-based routing:** the request is sent to the replica with the lowest recent latency. This can improve overall latency but may cause instability if all replicas converge on the same target and then overload.
- **Request-aware routing:** the load balancer considers request characteristics (input size, expected output size, priority) and routes accordingly. For example, long prompts might be routed to less loaded replicas to avoid blocking. This requires the load balancer to understand workload semantics.
- **Affinity-based routing:** related requests are routed to the same replica. This is useful when the replica caches expensive computation (such as a KV cache for shared prompt prefixes), because it increases cache hit rates. However, it can create load imbalance.

Load balancing is not purely a networking problem. In inference systems, the load balancer must be integrated with the serving system's scheduling and

resource management. A load balancer that distributes requests evenly across replicas is ineffective if one replica is running a memory-intensive model and another is running a small model. Effective load balancing requires visibility into replica state, not just connection counts.

For large-scale distributed inference, load balancing operates at multiple levels: between nodes, between GPUs within a node and within a serving process between different models or model instances. Each level may use different strategies optimized for its context.

Scaling Dimensions: Replication, Sharding, Partitioning

Inference systems scale in three primary dimensions: replication, sharding and partitioning. Each dimension addresses different constraints and introduces different trade-offs.

Replication is the simplest scaling strategy: run multiple identical copies of the model, each capable of serving any request. Replication increases throughput and improves availability. If one replica fails, others continue serving. Replication is also the basis for load balancing. The limitation of replication is that each replica requires its own copy of the model in memory, which can be expensive for large models. A 70-billion-parameter model in FP16 precision requires about 140 GB of memory. If you need ten replicas for throughput, you need 1.4 TB of GPU memory, spread across many GPUs. Replication is most effective when the model fits on a single device and you need more capacity.

Sharding splits the workload across replicas, with each replica handling a subset of requests based on some key. For example, a recommendation system might shard by user ID: users with IDs 0 to 999 are handled by replica 1, users with IDs 1000 to 1999 by replica 2 and so on. Sharding allows the system to scale beyond the capacity of a single replica without requiring every replica to hold a full copy of all data. However, sharding introduces complexity: requests must be routed correctly and re-balancing the shard distribution when replicas are added or removed requires data migration. Sharding is common in retrieval-based systems where each replica holds a subset of the index or data.

Partitioning splits the model itself across multiple devices or replicas. This is necessary when a single model is too large to fit on one device. There are several forms of model partitioning:

- **Tensor parallelism** partitions individual operations (such as matrix multiplications) across devices, with devices communicating during computation. Each device holds a slice of the model weights.
- **Pipeline parallelism** partitions layers of the model across devices, with each device processing its layers in sequence and passing activations forward. Each device holds a subset of the layers.
- **Data parallelism with model sharding** (sometimes called ZeRO or FSDP for training, with analogous inference variants) partitions model parameters across devices and redistributes them as needed during computation.

Model partitioning is powerful but complex. It adds communication overhead, requires careful load balancing across devices and introduces failure modes that do not exist in replicated systems. If one device in a partitioned model fails, the entire request must be retried or the model instance must be restarted. Partitioning is necessary for very large models but should not be used when replication would suffice, because replication is simpler and more resilient.

Choosing the right scaling dimension depends on the model size, the hardware and the traffic requirements. For small models with high traffic, replication is usually best. For large models with moderate traffic, partitioning may be necessary. For systems with massive data (such as search or recommendation), sharding is often essential. Many production systems use a combination: partitioning within a replica group, replication across groups and sharding across the overall system.

API Design for Inference: Synchronous, Asynchronous, Streaming, WebSockets

The API is the contract between the inference system and its clients. How the API is designed affects usability, performance, resource usage and client architecture.

Synchronous (blocking) APIs are the simplest: the client sends a request and waits for the response. The connection remains open until the response is complete. This is the standard pattern for most web APIs and works well for short-lived requests with predictable response times. The disadvantage is that the client blocks and if the request takes longer than expected, the

client must either wait indefinitely or implement its own timeout logic. For inference, synchronous APIs are appropriate for workloads where requests complete quickly and reliably.

Asynchronous (non-blocking) APIs decouple request submission from response retrieval. The client sends a request and receives an immediate acknowledgment with a request ID. The client then polls an endpoint or waits for a callback to retrieve the result. Asynchronous APIs are essential for long-running inference tasks where blocking would be impractical: large batch jobs, complex reasoning tasks, or video processing. They allow clients to continue other work while waiting. The trade-off is added complexity: clients must manage request IDs, implement polling or callback handlers and deal with eventual consistency between request submission and result availability.

Streaming APIs send the response incrementally as it becomes available rather than all at once. This is particularly important for LLM inference, where output is generated token by token. With streaming, the client begins receiving tokens as soon as the first token is generated, creating a responsive user experience even if the full response takes several seconds to complete. Streaming reduces perceived latency dramatically: users see output appearing in real-time rather than waiting for the entire response. Implementing streaming requires server-sent events (SSE), HTTP chunked transfer encoding, or WebSocket connections. The complexity is on both server and client: the server must support streaming and the client must handle partial responses and stream termination correctly.

WebSocket connections provide bidirectional, persistent communication between client and server. Unlike HTTP, which is request-response oriented, WebSockets maintain an open connection that can carry messages in both directions at any time. WebSockets are useful for interactive inference applications: real-time chat, collaborative editing, or continuous audio processing. They reduce connection overhead for frequent exchanges and enable the server to push updates to the client proactively. The trade-off is that WebSockets are harder to scale and manage than HTTP because they maintain stateful connections that must be tracked, load-balanced and recovered.

The choice of API pattern is not purely technical. It depends on user expectations, client capabilities and operational constraints. A consumer chat application almost certainly needs streaming. A backend fraud detection system probably uses synchronous calls. A batch data processing pipeline likely uses asynchronous APIs. Good API design exposes the right patterns for the

primary use cases without overcomplicating the interface for simpler cases.

Chapter 3: Computing Hardware for Inference

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Compute Paradigms: Tensor Operations, Matrix Multiplication, Sparsity

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

GPUs: Architecture, Memory Hierarchy, Streaming Multiprocessors, Tensor Cores

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

TPUs and ASIC Accelerators: Fixed-function vs Configurable

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

CPUs for Inference: When They Make Sense, Vector Instructions, Memory

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Memory Hierarchy and Bandwidth: DRAM, HBM, SRAM, PCIe

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Interconnects: NVLink, InfiniBand, Ethernet, NUMA Effects

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Hardware Cost Models: CapEx, OpEx, Power, Cooling

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Chapter 4: Model Serving Systems and Runtimes

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

What a Serving System Does: Beyond “Loading a Model”

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Request Lifecycle: From HTTP to Model Execution to Response

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Model Loading, Initialization and Warmup

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Inference Engines: TorchServe, Triton, vLLM, TensorRT Serving, ONNX Runtime and Others

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Runtime Responsibilities: Memory Management, Scheduling, Optimization

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Containerization and Deployment: Docker, Kubernetes Integration

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Choosing a Serving System: Evaluation Criteria

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Chapter 5: Concurrency, Batching and Scheduling

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Why Batching Works: Amortizing Overhead and Exploiting Parallelism

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Static vs Dynamic Batching: Trade-offs and Implementation

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Continuous Batching: Mechanism, Benefits and Limitations

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Queueing Theory Basics for Inference: Arrival, Service, Waiting

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Scheduling Algorithms: FIFO, Priority, Fairness, ML-Specific Scheduling

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Concurrency Control and Thread Pool Design

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Admission Control and Load Shedding Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Overlapping Computation and Communication

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Chapter 6: Memory Management and KV Cache Systems

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Memory in Inference: Model Weights, Activations, KV Cache, Temporaries

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

The KV Cache: Purpose, Growth, Memory Implications

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Paged Attention: How It Works, Memory Savings, Implementation

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Memory Fragmentation and Compaction

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

GPU Memory Management: Allocations, Caching, Pinning

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Spilling KV Cache to Host Memory: Trade-offs

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Memory-Aware Batching and Eviction Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Chapter 7: Model Optimization: Quantization

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Why Quantization Works: Redundancy in Learned Parameters

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Fixed-Point Arithmetic: Formats, Ranges, Scaling Factors

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Post-Training Quantization: PTQ Methods and Accuracy

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Quantization-Aware Training: Mechanism and When to Use It

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Per-Tensor, Per-Channel, Per-Token Quantization

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Mixed Precision Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Hardware Support and Compatibility

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Quantization Impact: Accuracy, Latency, Memory, Power

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Common Pitfalls and Debugging Quantization Issues

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Chapter 8: Model Optimization: Pruning, Distillation and Compression

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Pruning: Motivation, Mechanisms and Types (Unstructured, Structured, Layer)

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Pruning Algorithms: Magnitude-Based, Gradient-Based, Iterative Methods

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Knowledge Distillation: Teacher-Student Framework, Loss Functions

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Pruning and Distillation Combined: Synergistic Benefits

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Compression Formats and Packaging

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Structured Sparsity and Hardware Acceleration

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Accuracy Trade-offs and Calibration

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

When Optimization Is Worth the Effort

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Chapter 9: Compilation and Graph Optimization

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Computation Graphs and Their Optimization Opportunities

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Graph Transformations: Constant Folding, Dead Code Elimination, Type Canonicalization

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Operator Fusion: Mechanism, Patterns, Performance Impact

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Memory Planning and Optimization: Allocation, Reuse, Graph Coloring

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Kernel Selection and Specialization

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Hardware-Aware Compilation: Loop Tiling, Blocking, Vectorization

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

End-to-End Compilers: TVM, MLIR, XLA, TensorRT

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Compilation Time vs Inference Gain: The Trade-off

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Chapter 10: Kernel-Level and Runtime Optimization

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Understanding Compute Kernels: What They Do, How They Execute

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Memory Access Patterns: Coalescing, Bank Conflicts, Cache Locality

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Cache Optimization: Register, L1, L2, Shared Memory

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Custom Kernel Development: When It Is Worth It

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

CUDA and GPU-Specific Optimization Techniques

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

CPU-Specific Optimizations: AVX, AMX, Loop Unrolling

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Runtime Optimizations: JIT Compilation, Caching, Warmup

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Profiling at the Kernel Level

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Chapter 11: Performance Engineering Methodology

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

The Performance Engineering Loop: Measure, Profile, Hypothesize, Optimize, Validate

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Establishing a Measurable Baseline

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Profiling Tools and Techniques: System, Runtime, Model, Kernel Levels

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Identifying Bottlenecks: Compute, Memory, Network, Queue

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

The Roofline Model for Inference

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Benchmarking Correctly: Workload Representation, Warmup, Statistical Validity

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Avoiding Benchmark Pitfalls and Misleading Measurements

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Regression Detection and Continuous Performance Testing

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Capacity Modeling and Saturation Analysis

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Chapter 12: Distributed Inference and Parallelism Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Why Distribute Inference: Memory, Throughput, Latency Requirements

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Data Parallelism for Inference: Replicated Models, Load Distribution

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Tensor Parallelism: Mechanism, Communication Patterns, Use Cases

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Pipeline Parallelism: Stages, Bubbles, Throughput Implications

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Hybrid Parallelism: Combining Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Distributed Communication: All-Reduce, All-Gather, Point-to-Point

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

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/apracticalguidetoinferenceengineering>.

Scaling Laws for Inference: Where Scaling Stops Helping

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

When Distributed Inference Is Overkill

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Chapter 13: LLM Inference: Prefill, Decode and Beyond

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Autoregressive Generation: Mechanics and Asymmetry

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Prefill and Decode: Two Distinct Workloads

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Attention Complexity: Quadratic Scaling and Its Consequences

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Efficient Attention Mechanisms: FlashAttention, Linear Attention, Sparse Attention

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Speculative Decoding: How It Works, Speedup Analysis, Error Handling

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Long-Context Inference: Challenges, Strategies, Trade-offs

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Multi-Turn Conversations and Session Management

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

LLM-Specific Metrics and Monitoring

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Chapter 14: Advanced LLM Optimization Techniques

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Prefix and Prompt Caching: Mechanism, Hit Rates, Memory Costs

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Chunked Prefill: Balancing Latency and Throughput

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Early Exit: Confidence-Based Premature Termination

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Layer Skipping and Adaptive Computation

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Request-Level Optimization: Priority, Slicing, Merging

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Context Window Optimization: Summarization, Retrieval, Sliding Window

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Tokenization Efficiency and Vocabulary Trade-offs

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Multi-Model Systems: Routing, Cascading, Speculative Hierarchies

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Chapter 15: Inference for Different Workload Classes

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Computer Vision Inference: Image Models, Batching Patterns, Hardware Fit

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Speech and Audio Inference: Streaming, Chunking, Real-Time Constraints

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Recommendation and Ranking: Scale, Low Latency, Batch Scoring

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Embedding Inference: High Throughput, Dimensionality, Caching

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Multimodal Inference: Composed Workflows, Asynchronous Stages

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Conventional ML: Trees, Linear Models, XGBoost, LightGBM

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Generalizable Principles vs Workload-Specific Techniques

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Polyglot Inference Platforms

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Chapter 16: Production Operations: Observability, SLOs and Reliability

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Defining SLOs and SLIs for Inference: Latency, Availability, Quality

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Metrics That Matter: What to Collect and Why

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Logging for Inference: Request-Level Observability

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Distributed Tracing for Inference Systems

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Alerting: Signal vs Noise, Smart Alerting Policies

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Health Checks and Self-Diagnostics

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Fault Isolation and Blast Radius Control

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Graceful Degradation Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Incident Response for Inference Systems

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Chapter 17: Deployment, Lifecycle Management and Versioning

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Model Versioning and Identification

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Deployment Strategies: Rolling, Blue-Green, Canary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

A/B Testing and Shadow Deployment for Models

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Rollback and Rapid Recovery

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Model Registry and Lifecycle Tracking

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Configuration Management for Inference

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Automated Testing: Functional, Performance, Drift

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Security: Authentication, Authorization, Isolation, Data Protection

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Abuse Prevention and Rate Limiting

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Chapter 18: Cost and Infrastructure Efficiency

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Cost Drivers in Inference: Hardware, Memory, Compute, Network, Energy

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Cloud vs On-Prem vs Hybrid: Cost and Control Trade-offs

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Instance Selection and Right-Sizing

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Autoscaling Policies: Metrics, Thresholds, Hysteresis

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Reserved, Spot and On-Demand Capacity Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Utilization Optimization: Packing, Sharing, Overcommitting

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Inference Cost Modeling: Per-Request, Per-Token, Per-User

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Energy Efficiency and Environmental Considerations

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Total Cost of Ownership: Beyond Infrastructure Bills

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Quality-Cost Trade-offs: When to Spend More for Better Performance

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Chapter 19: End-to-End System Design: Realistic Scenarios

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Design Process: From Requirements to Architecture

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Scenario 1: High-Traffic Chat API with Strict Latency SLOs

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Scenario 2: Batch Embedding Pipeline for Search Indexing

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Scenario 3: Real-Time Recommendation System at Scale

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Scenario 4: Multi-Model Platform Serving Heterogeneous Workloads

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Capacity Estimation Walkthrough: Step by Step

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Hardware Selection Framework

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Optimization Strategy Selection Decision Tree

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Monitoring and Continuous Improvement

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Common Architectural Anti-Patterns and How to Avoid Them

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Conclusion: Principles, Patterns and the Future

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Enduring Principles of Inference Engineering

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Patterns That Recur Across Systems

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

Emerging Trends: New Hardware, New Models, New Techniques

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

What to Learn Next

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

The Future of Inference: Speculation Grounded in Evidence

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.

References

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/apracticalguidetoinferenceengineering>.