How to become a Staff AI Engineer

Chapter 21: LLM Inference Engineering for AI Engineering

Celso Sousa
LinkedIn: https://www.linkedin.com/in/celso-sousa/?locale=en-US
Portfolio: https://celsosousa.com.br/project-portfolio/
Ebook: https://celsosousa.com.br/ebook-staff-ai-engineer/

Chapter Objective

Training a Large Language Model is expensive. However, transforming a trained model into a service capable of operating continuously in production, under heterogeneous workloads, concurrency spikes, and stringent latency requirements, can represent an even more complex engineering problem.

LLM Inference Engineering addresses the transformation of a model into an operational capability capable of serving real-world workloads efficiently, predictably, and reliably. The problem is not merely executing forward passes on a GPU, but coordinating compute, memory, KV cache, batching, scheduling, communication, replicas, and admission policies within the constraints imposed by the product.

A production inference system typically needs to balance several dimensions simultaneously:

- **Low latency improves the user experience:** However, aggressive policies designed to minimize individual waiting time can reduce batching efficiency and throughput;
- **High throughput increases the amount of work performed per unit of hardware:** However, maximizing tokens per second does not necessarily deliver the best TTFT or the best interactive experience;
- **Higher GPU utilization can improve economics:** But operating continuously near saturation reduces headroom and can increase queueing and tail latency;
- **Higher concurrency increases serving capacity:** However, it increases pressure on KV cache, memory bandwidth, and the scheduler;
- **A larger context window increases the amount of information available to the model:** However, it consumes more compute during prefill and more memory throughout the lifetime of the sequence;
- **Tenant isolation protects fairness and capacity:** But it requires quotas, admission control, and scheduling policies compatible with product tiers;
- **Reliability keeps the service available in the presence of failures:** However, retries, additional replicas, and fallbacks also introduce infrastructure costs and operational complexity.

The difficulty increases because LLM requests are not homogeneous units like traditional API calls. A request may contain a few dozen or tens of thousands of input tokens, generate a handful of tokens or thousands of them, use different model tiers, have different priorities, and require streaming or not.

In addition, execution has two phases with different computational profiles:

- **Prefill processes the initial context:** It offers greater parallelism, builds the KV cache, and can be compute-intensive;
- **Decode generates new tokens autoregressively:** It has sequential dependencies, reuses the KV cache, and tends to be highly sensitive to memory bandwidth;
- **Optimizing prefill does not mean optimizing decode:** A configuration may produce excellent TTFT while still exhibiting inadequate TPOT;
- **KV cache avoids recomputation:** At the same time, it can become one of the primary consumers of memory and limit concurrency;
- **Batching improves hardware utilization:** However, waiting for additional requests to form batches can increase latency;
- **Continuous batching adapts the batch to the workload:** Requests can enter and leave dynamically as sequences complete;
- **Quantization reduces memory footprint and bandwidth pressure:** The benefit must be evaluated against potential quality regressions;
- **Speculative decoding trades auxiliary compute for lower decode latency:** The gain depends on the acceptance rate of proposed tokens;
- **Model parallelism makes it possible to serve models that exceed the limits of a single device:** However, it introduces communication overhead into the critical path;
- **Autoscaling must account for work in tokens:** Request count alone poorly represents the heterogeneity of LLM workloads;
- **Inference economics must be tied to completed tasks:** GPU utilization in isolation does not represent economic value.

For a Staff AI Engineer, therefore, the relevant question is not simply which serving library to use or which GPU offers the highest throughput in a benchmark. The more important question is: **which resource limits serving for this workload (compute, memory bandwidth, KV cache, scheduler, network, or model choice) and which optimization improves the critical path without degrading quality, reliability, or unit economics?**

This mental model will guide the remainder of the chapter.

21.1. Mental Model of LLM Inference

An LLM request can be decomposed, in simplified form, into two major phases: prefill and decode. This separation is essential because each phase has different parallelism patterns, resource consumption characteristics, and effects on user-perceived metrics.

21.1.1. Prefill

During prefill, the model processes the tokens in the initial context and produces the representations required to begin generation. This phase must perform several fundamental operations:

- **Process the entire prompt:** Input tokens must traverse the model layers;
- **Build the KV cache:** Keys and Values associated with the context must be preserved to avoid recomputation during decode;
- **Produce the first token:** Completion of prefill enables autoregressive generation to begin.

For this reason, prefill directly affects Time to First Token, or TTFT. Its cost depends primarily on several workload characteristics:

- **Larger prompts increase compute:** More tokens must be processed through the layers before the response can begin;
- **Larger models increase cost per token:** More parameters and operations must be executed;
- **Batch size changes hardware efficiency:** Grouping prompts can increase utilization, but can also introduce waiting time;
- **Long context can dominate the latency budget:** In applications with extensive contexts, the cost incurred before the first token may substantially exceed the cost observed in small requests;
- **Attention over larger contexts increases computational work:** Interaction with an increasing number of tokens can make prefill significantly more expensive;
- **Prompt structure affects economics:** Reprocessing large system prompts, policies, or schemas across thousands of requests creates repetitive compute.

This means context engineering is also performance engineering. Reducing redundant information in the context can lower TTFT, increase throughput, and reduce cost without changing the model or hardware.

In enterprise workloads, therefore, increasing the context window should not be treated as a free optimization. The additional context must produce enough improvement in task success to justify the additional compute and memory consumption.

21.1.2. Decode

After the context has been processed, generation enters the decode phase. At this stage, new tokens are produced autoregressively, and each token depends on the sequence generated so far.

Each decode step must execute a cycle similar to the following:

1. Process the most recent token;
2. Consult the state preserved in the KV cache;
3. Produce the probability distribution for the next token;
4. Select the token;
5. Repeat the process.

This sequential dependency creates a profile very different from prefill:

- **Parallelism within a single sequence is limited:** The next token depends on the previous result;
- **KV cache eliminates recomputation of the previous context:** However, the cache must be accessed continuously;
- **Operations repeat for every new token:** Long outputs keep the sequence active for longer;
- **Decode tends to be sensitive to memory bandwidth:** Weights and KV cache must be accessed repeatedly;
- **Concurrency increases memory pressure:** Each active sequence has its own state;
- **Streaming reduces perceived latency:** The user can begin consuming the response while generation continues.

As a consequence, output length is an important architectural and economic variable. If an application can reduce the average number of output tokens by half without degrading delivered value, it can simultaneously improve latency, throughput, and cost without changing the model. This is an important example of system optimization: **some of the best inference optimizations happen outside the runtime.**

21.1.3. TTFT, TPOT, and End-to-End Latency

The generic statement that “the model is slow” does not provide enough information for diagnosis. LLM latency must be decomposed because different portions of total execution time have different causes and optimization mechanisms.

The primary metrics include:

- **TTFT - Time to First Token:** Measures how long the user waits before beginning to receive the response;
- **TPOT - Time per Output Token:** Measures the interval between tokens during decode;
- **Total latency:** Measures the complete time between request arrival and generation completion;
- **Queue time:** Measures how long the request remained waiting for capacity before execution.

Each metric points to different classes of problems. High TTFT can originate from different causes:

- **Long prompts increase prefill:** A larger number of input tokens must be processed;
- **Queueing increases time before execution:** The model itself may be fast while the system is saturated;
- **Slow prefill indicates a problem in the context-processing phase:** Compute, batching, or context size may be limiting execution.

High TPOT typically suggests another class of bottlenecks:

- **Decode may be the bottleneck:** Token-by-token generation is limiting the response;
- **Memory bandwidth pressure can reduce speed:** Data must be moved continuously during generation;
- **Excessive concurrency can degrade each individual sequence:** Many sequences compete for the same resources.

Total latency, by contrast, may simply result from an extremely long response. A system with adequate TTFT and TPOT can still exhibit high completion time if it generates thousands of tokens. For this reason, the first step in any performance investigation should be to identify which phase is degraded. Without this decomposition, local optimizations may target the wrong resource and increase complexity without improving the user experience.

21.2. KV Cache, Memory, and Scheduling

Efficient autoregressive inference depends heavily on the ability to preserve and manage state. In this context, KV cache stops being an implementation detail and becomes a first-class resource for capacity planning.

21.2.1. KV Cache

KV cache preserves the Keys and Values produced by previous tokens, preventing the entire context from being recomputed during each new decode step. Without this mechanism, autoregressive generation would be significantly more expensive.

However, KV cache consumption grows along several dimensions:

- **More concurrent sequences increase memory consumption:** Each active sequence must preserve its state;
- **Greater context length increases the initial cache size:** More tokens must remain represented;
- **Greater output length extends and expands state:** The sequence continues to grow as tokens are generated;
- **Model dimensions affect the footprint per token:** Larger architectures may require larger representations;

- **More layers increase the amount of preserved state:** The cache must retain information associated with different layers.

The practical consequence is that a model may comfortably fit in GPU memory in terms of weights while still being unable to support the required concurrency. The limiting resource may not be space for the model, but the space remaining for sequences.

21.2.2. Capacity and Memory Pressure

Capacity planning must decompose available memory across its primary consumers:

- **Model weights consume a relatively stable portion:** Their size depends on the model and the representation being used;
- **KV cache grows dynamically with the workload:** Context, output, and concurrency change its consumption;
- **Runtime buffers also require headroom:** Kernels and internal operations require additional memory.

When KV cache consumes the remaining memory, new requests cannot be admitted regardless of whether compute remains available. This creates important trade-offs:

- **Context length vs. concurrency:** Longer sequences reduce the number of simultaneous requests that can remain active;
- **Output length vs. capacity:** Long generations keep resources occupied for longer;
- **Larger model vs. cache headroom:** More memory reserved for weights means less space available for sequences;
- **Precision vs. memory:** Smaller representations can increase capacity and enable greater density.

For this reason, concurrency should not be planned only as a "number of requests." A more useful approximation needs to consider: **active sequences × token footprint.** This reasoning explains why two services with the same RPS can require completely different infrastructures when their context-length and output-length distributions differ.

21.2.3. KV Cache Management

Beyond raw capacity, the runtime must manage memory dynamically. Real-world workloads contain sequences of different lengths that begin and end at different times.

This creates operational problems such as:

- **Fragmentation reduces effectively usable memory:** Free regions may exist but may not be organized in a useful way for new sequences;
- **Uneven sequence lengths complicate allocation:** Short and extremely long requests coexist in the same pool;
- **Memory churn increases management overhead:** Allocations and deallocations occur continuously as the workload evolves.

A naive strategy that reserves large contiguous memory regions per sequence can waste a significant portion of VRAM. Modern runtimes can use memory blocks or pages to improve efficiency:

- **Granular allocation reduces fragmentation:** Smaller regions can be reused more flexibly;
- **Greater effective capacity increases concurrency:** More sequences can remain active simultaneously;

- **Heterogeneous workloads are accommodated more effectively:** Short and long requests no longer require equivalent rigid reservations.

In high-concurrency inference, therefore, the memory allocator and scheduler can be as relevant to performance as the kernels used by the model. A common mistake is to evaluate only available FLOPS while ignoring how sequence state will be materialized and moved during real operation.

21.2.4. Prefix Caching

Many requests share large identical prefixes. This frequently occurs in enterprise applications, where system instructions, policies, tool schemas, and templates are reused across nearly every interaction.

Common examples include:

- **System instructions remain stable across requests:** Reprocessing them continuously represents repeated work;
- **Policies may be shared by an application or tenant:** Their prefill cost can be amortized;
- **Tool schemas may contain hundreds or thousands of tokens:** Agents frequently resend these structures in every interaction;
- **Long templates create predictable prefixes:** Invariant portions can be reused.

Prefix caching allows work associated with a previously processed prefix to be reused. The benefits are significant:

- **TTFT can decrease:** Part of prefill no longer needs to be executed again;
- **Compute cost is reduced:** Identical tokens do not need to traverse all layers repeatedly;
- **Throughput can increase:** The GPU becomes available to process genuinely new work.

However, cache reuse is correct only when the prefix remains compatible. Changes to some components may require invalidation:

- **A different model version invalidates previous representations:** State depends on the parameters used;
- **A different prompt version changes the prefix:** An apparently small modification may make the cache incompatible;
- **A different tokenizer changes the token sequence:** Textual equality does not guarantee equality of representation;
- **A different tool schema changes the effective input:** Tool modifications are also part of the context.

Prefix caching, therefore, is not merely a performance mechanism. It requires explicit versioning and structural compatibility to prevent incorrect state reuse.

21.2.5. Static, Dynamic, and Continuous Batching

Batching combines multiple requests to increase GPU utilization. However, the efficiency of the strategy depends heavily on workload distribution. Static batching waits for a batch to form before executing it.

This strategy works best when requests have similar characteristics:

- **Similar sizes reduce waste:** Sequences contain comparable amounts of work;
- **Predictable arrival patterns make batch formation easier:** The runtime can gather requests without excessive waiting;
- **Similar durations reduce idle time:** Sequences tend to complete at approximately the same time.

The problem is that real-world LLM workloads are rarely homogeneous. One request may generate twenty tokens while another generates two thousand. If both are bound to a rigid batch, the shorter sequence may become constrained by the behavior of the longer one.

Continuous batching addresses this problem by allowing requests to enter and leave the active set dynamically. When a sequence completes, another can immediately occupy the released capacity.

This produces several important benefits:

- **Fewer slots remain idle:** Released capacity can be reused immediately;
- **Heterogeneous outputs are handled more efficiently:** The batch does not need to wait for the longest sequence;
- **Effective concurrency increases:** The scheduler can keep hardware busy with available work;
- **Queue efficiency improves:** New requests can be incorporated into the execution flow without waiting for entirely new batches.

However, continuous batching transfers part of the complexity to the scheduler. The scheduler must then balance competing objectives:

- **Fairness prevents starvation:** No workload class should remain indefinitely without progress;
- **TTFT favors newly arrived requests:** Interactive applications value a fast start;
- **Decode throughput favors already-active sequences:** Excessive interruption of progress can degrade tokens per second;
- **Utilization favors dense batches:** However, maximizing density can increase queue time.

Modern LLM serving is therefore, to a large extent, an online scheduling problem involving heterogeneous sequences.

21.2.6. Scheduling and Head-of-Line Blocking

When multiple requests compete for the same model pool, the scheduler must determine which requests are admitted, which remain queued, and which sequences advance at any given time. An inadequate policy can create head-of-line blocking, where heavy requests delay much smaller requests.

The scheduler can consider multiple signals:

- **Arrival time preserves a simple form of fairness:** Older requests may receive preference;
- **Priority represents criticality:** Interactive or operationally critical traffic may advance first;
- **Token budget approximates the amount of work:** Requests with large contexts or outputs may be handled differently;
- **Tenant tier represents commercial priorities:** Premium and standard products may have different SLOs.

These policies introduce additional trade-offs:

- **Fairness vs. efficiency:** Always prioritizing small requests can increase throughput but cause starvation for long-running jobs;
- **Interactive vs. batch workloads:** Human users generally require lower latency than offline workloads;
- **Premium vs. standard tenants:** Different service levels may require different priorities;
- **Throughput vs. user experience:** A scheduler that maximizes tokens per second may not minimize user-perceived time.

The scheduler should reflect product priorities, not merely maximize a hardware metric. This represents an important shift in perspective: **scheduling policy is also product policy.**

21.3. Inference Optimizations

After identifying the limiting resource, different techniques can be used to reduce memory footprint, bandwidth pressure, or the amount of work executed. None of these techniques should be applied independently of measurement and evaluation.

21.3.1. Quantization

Quantization represents weights and, in some architectures, other states using lower precision. Common formats include:

- **INT8 reduces representation size relative to higher precisions:** It can provide meaningful memory savings with limited degradation in certain workloads;
- **INT4 further reduces footprint:** However, it increases the risk of regressions;
- **Weight-only quantization primarily reduces weight representation:** Other operations may remain at higher precision.

The potential benefits are significant:

- **Smaller memory footprint allows larger models to run on fewer GPUs:** This can reduce hardware requirements;
- **Lower memory bandwidth pressure reduces the number of bytes moved:** Decode can benefit when bandwidth is the limiting resource;
- **Higher potential concurrency leaves more space for KV cache:** Reducing weight footprint can increase the number of active sequences;
- **Lower infrastructure cost improves unit economics:** The same traffic may require fewer resources.

The primary trade-off is model quality. More aggressive quantization may affect different types of behavior:

- **Reasoning may regress:** Particularly in tasks that are more sensitive to numerical representation;
- **Rare-token behavior may degrade:** Infrequent cases can conceal losses that are not visible in averages;
- **Long-tail tasks can be disproportionately affected:** A small average loss does not mean every segment remains stable.

For this reason, quantization must be evaluated on the real workload using a representative regression suite. Aggregate metrics can hide degradation precisely in the most critical cases.

21.3.2. Weight-Only Quantization

In certain workloads, weights account for a large portion of memory and bandwidth consumption. Weight-only quantization reduces the representation of those weights while selected operations continue to use higher precision.

This strategy can produce an attractive balance:

- **Memory savings reduce footprint:** More capacity can remain available for KV cache or replicas;

- **Model quality can remain close to the baseline:** Precision reduction does not necessarily need to apply to every state and operation.

However, reducing the stored model size does not guarantee faster execution. The hardware and runtime must provide efficient kernels for the selected format. Otherwise, conversions or poorly optimized operations can eliminate part of the benefit.

The relevant principle is: **storage compression does not automatically imply execution acceleration.** The gain must be measured through end-to-end latency, TPOT, throughput, and capacity, rather than inferred solely from checkpoint size.

21.3.3. Speculative Decoding

Autoregressive decode has sequential dependencies. Speculative decoding attempts to reduce this bottleneck by using a smaller model or another cheaper mechanism to propose multiple tokens in advance.

The target model verifies these proposals. When multiple tokens are accepted, more than one token can be confirmed with fewer full cycles of the primary model.

Potential gains include:

- **Lower decode latency:** Fewer full passes through the target model may be required;
- **Higher tokens per second:** A high acceptance rate allows multiple tokens to advance per cycle;
- **Quality can remain equivalent to the target-model distribution:** Provided that the verification mechanism preserves the desired behavior.

However, speculation introduces auxiliary work. If a large fraction of proposed tokens are rejected:

- **Draft compute is wasted:** The system performs work that does not reduce the required number of target-model passes;
- **Verification overhead remains:** The target model still needs to validate proposals;
- **The net benefit can disappear:** The architecture becomes more complex without producing a latency gain.

The fundamental economic condition can be summarized as: **draft cost + verification cost < decode cost saved.** Speculative decoding should therefore be applied when decode truly represents the bottleneck and when the acceptance rate remains sufficiently high on the real workload.

21.3.4. Model Parallelism

Large models may exceed the memory or compute capacity of a single GPU. In this case, weights and operations must be distributed across multiple devices.

The primary strategies include tensor parallelism and pipeline parallelism. Tensor Parallelism divides the same layer across different GPUs.

The primary benefits are:

- **Memory footprint per GPU is reduced:** Weights no longer reside entirely on a single device;
- **Compute is distributed:** Multiple GPUs participate in executing the same layer.

However, communication then occurs frequently between devices. Tensor Parallelism works best when GPUs have high-bandwidth, low-latency interconnects. Otherwise, communication overhead can consume a significant portion of the gain obtained through distribution.

Pipeline Parallelism distributes groups of layers across different devices. This strategy can be useful for very deep models but creates other challenges:

- **Each stage must remain occupied:** Idle stages reduce global efficiency;
- **Pipeline bubbles reduce utilization:** Imbalance or insufficient work leaves resources inactive;
- **Requests must traverse stages sequentially:** In interactive inference, this can increase latency.

For this reason, throughput and latency must be evaluated jointly.

Model parallelism should be used to solve a concrete memory or compute constraint. Distributing a model simply because multiple GPUs are available can add communication without producing a net benefit.

21.4. Scale-Out, Autoscaling, and Model Routing

When a single replica cannot support the workload, inference begins to involve scale-out. At that point, topology, load balancing, replica placement, startup time, and routing become part of the critical path.

21.4.1. Multi-GPU and Multi-Node Serving

When serving extends beyond the boundaries of a single server, the network begins to participate directly in execution. Several factors become particularly important:

- **Interconnect bandwidth limits data transfer between devices:** Communication-intensive strategies depend on fast links;
- **Network latency can enter the critical path:** Frequent operations across nodes amplify small latencies;
- **Topology influences placement:** Not every GPU and node combination has the same physical proximity.

A configuration that performs very well across GPUs in the same server can behave very differently when the same ranks are distributed across machines. Placement must consider communication frequency. Components that constantly exchange data should remain on faster links. Tensor-parallel ranks, for example, generally benefit from greater physical proximity.

Multi-node serving also expands the failure surface:

- **Failure of one node can take down an entire replica:** Distributed dependencies create new failure modes;
- **Replication must consider blast radius:** Capacity should not depend on a single fragile composition;
- **Health checking must detect degradation quickly:** Problematic workers should not continue receiving traffic;
- **Failover must preserve availability:** The system needs a strategy for component loss.

Scale-out, therefore, should not be analyzed only as an increase in capacity. It changes reliability and operational complexity.

21.4.2. Serving Replicas and Load Balancing

Scale-out typically uses multiple replicas of the model. The load balancer must distribute work across these pools in a manner compatible with request heterogeneity.

Simple round-robin can be inadequate. Two replicas can have exactly the same number of requests while processing completely different workloads:

- **One replica may process several long-context requests:** Each request represents a large amount of work;
- **Another may process short queries:** The same request count results in significantly lower load.

For this reason, load balancing can consider more representative signals:

- **Active tokens approximate work currently in execution:** They better capture differences in request size;
- **Queue length reveals local saturation:** A replica with a larger queue may be overloaded;
- **KV cache pressure indicates the ability to admit new sequences:** Available compute does not imply available memory;
- **Estimated work can predict request cost:** Context length and other characteristics can guide routing.

Balancing LLM serving solely by request count can produce hotspots even when traffic appears uniformly distributed. The appropriate unit of work must reflect tokens and state, not merely connections.

21.4.3. Autoscaling

Traditional autoscaling based only on CPU utilization or request count poorly represents LLM workloads. More useful signals can include:

- **Queue depth indicates that demand is exceeding available capacity:** Persistent queue growth is a direct signal of saturation;
- **Active sequences show effective concurrency:** Requests waiting or already completed should not be confused with work currently executing;
- **Tokens in flight approximate the amount of work:** Sequences have different sizes;
- **GPU utilization indicates compute consumption:** However, it must be analyzed together with other metrics;
- **KV cache occupancy reveals memory pressure:** Capacity can be exhausted before compute reaches 100%.

The primary challenge is startup time. Loading a large model may take long enough to make reactive autoscaling inadequate for sudden spikes. By the time a new replica becomes operational, the queue may already have degraded the SLOs.

Several strategies help manage this problem:

- **Warm capacity reduces response time:** Pre-provisioned resources absorb sudden demand growth;
- **Predictive scaling anticipates known patterns:** Seasonal traffic can enable provisioning ahead of the peak;
- **Minimum replica pools preserve baseline capacity:** The system avoids starting from an excessively small configuration.

These strategies create a direct economic trade-off:

- **Excess capacity reduces latency:** However, it increases idle GPU cost;
- **Minimal capacity reduces idle cost:** But it increases queueing when demand grows.

Inference autoscaling is therefore simultaneously an SRE and FinOps problem. The architecture must decide how much it costs to maintain headroom and how much it costs to violate SLOs when that headroom does not exist.

21.4.4. Model Routing and Tiered Serving

Not every request needs to use the same model. A platform can operate different model tiers and route workloads according to their complexity and requirements.

For example, the serving layer may include:

- **Small model for simple and frequent tasks:** Reduces cost and latency;
- **Medium model for intermediate tasks:** Provides a balance between capability and economics;
- **Advanced model for complex problems:** Reserves frontier capability for requests that actually require it.

Routing can use different signals:

- **Task type identifies known classes of work:** Some tasks can be assigned directly to smaller models;
- **Complexity estimates difficulty:** More complex requests can be escalated;
- **Latency budget influences selection:** Interactive products may favor smaller models when possible;
- **Tenant tier can define access to different capabilities:** The model used can be part of the commercial offering;
- **Confidence can guide escalation:** Low confidence at a lower tier can justify rerunning the request on a more capable model.

The benefits include:

- **Lower average cost:** Simple cases stop consuming the most expensive model;
- **Higher capacity:** Workload is distributed across different pools;
- **Better latency for simple tasks:** Requests no longer compete unnecessarily with larger models.

However, the router then becomes part of the quality pipeline:

- **Under-routing reduces quality:** A complex task may be assigned to a model that cannot solve it;
- **Over-routing wastes resources:** Simple tasks continue consuming expensive capacity;
- **Routing requires its own evaluation:** Incorrect router decisions can degrade the system even when all models themselves function correctly.

Model routing is one of the primary mechanisms for improving quality per dollar. The relevant decision is not to select “the best model”, but to allocate capability proportional to the difficulty of each request.

21.5. Streaming, Reliability, Observability, and Security

A serving layer cannot be evaluated only by the speed of the happy path. Production systems must handle perceived latency, partial failures, saturation, degradation, abuse, and diagnostic requirements.

21.5.1. Streaming and User-Perceived Latency

Streaming sends tokens to the user as they are generated. This does not necessarily reduce total completion time. However, it significantly changes the experience because the user can begin consuming the response before generation completes.

The primary benefits are:

- **Better UX:** The application appears to respond more quickly;
- **Early feedback:** The user can see that the request is making progress;
- **Long outputs become more tolerable:** The entire wait no longer occurs before any content is presented.

Streaming also creates new challenges:

- **Clients must process partial outputs:** The interface must handle incomplete content;
- **Errors can occur after tokens have already been displayed:** Late failures can no longer be fully hidden;
- **Safety and post-processing become more difficult:** Validating the entire response before displaying it may be incompatible with immediate streaming.

This highlights an important distinction: **perceived latency and system latency are not the same metric.** Product architecture must account for both.

21.5.2. Reliability and Fallbacks

Model serving must assume that failures will occur. The problem is not merely preventing failures, but controlling their effects.

Failures can occur in different components:

- **A GPU can fail or degrade:** Hardware should not be treated as infallible;
- **The runtime can enter an invalid state:** Bugs and resource exhaustion can affect workers;
- **A model replica can become unavailable:** A single serving unit cannot represent the entire capacity;
- **An external provider can fail:** Managed dependencies also experience outages and limits.

A reliability strategy can combine different mechanisms:

- **Health checks remove problematic replicas:** Requests should not continue to be routed to degraded workers;
- **Timeouts bound stuck requests:** A stalled inference should not consume resources indefinitely;
- **Retries must be controlled:** Rerunning long prompts can duplicate cost and worsen congestion;
- **Fallback models preserve availability:** An alternative tier can serve part of the traffic when the primary model is unavailable;
- **Graceful degradation preserves partial value:** Certain capabilities can be reduced instead of causing the entire product to fail.

However, fallback behavior must be evaluated semantically. An endpoint that returns HTTP 200 using a significantly inferior model may be technically available while being functionally degraded.

Availability for an AI system should not be measured only by its ability to return responses. It is also necessary to determine whether the response preserves the minimum quality level required by the product.

21.5.3. Observability

LLM inference requires specialized telemetry. Traditional infrastructure metrics remain relevant, but they are insufficient to explain the behavior of the serving layer.

Several metrics are particularly important:

- **TTFT diagnoses initial waiting time:** It helps separate queue and prefill problems;
- **TPOT reveals decode performance:** It enables identification of degradation during generation;
- **Input tokens explain prefill cost:** Context distributions reveal changes in workload;
- **Output tokens explain generation time:** Growing outputs can degrade latency and economics;
- **Tokens per second measure throughput:** This is more representative of work than request count alone;
- **Active sequences approximate real concurrency:** They show the number of states executing simultaneously;
- **KV cache utilization reveals memory pressure:** It helps anticipate capacity limits;
- **Queue time separates saturation from model latency:** The model may be healthy while the queue grows;
- **GPU utilization shows compute usage:** It should be interpreted together with other resources;
- **GPU memory utilization identifies capacity limits:** It can expose pressure before OOM occurs;
- **Effective batch size reveals scheduler efficiency:** Configured batches and batches actually executed may differ;
- **Speculative acceptance rate reveals the true benefit of speculation:** Low acceptance can indicate that the optimization is introducing overhead.

Global averages are rarely sufficient. Telemetry must also be segmented along relevant dimensions:

- **Model:** Different models have different latency and capacity profiles;
- **Tenant:** A specific customer may generate a disproportionate workload;
- **Endpoint:** Different features may have different distributions;
- **Context-length bucket:** Long-context requests can conceal tail latency when mixed with small queries.

This segmentation makes it possible to locate the origin of the problem rather than merely observing that the global average has degraded. Quality observability allows the system to answer “Which component degraded?”. Inference observability must allow it to answer: **which resource is saturated, and for which workload class?**

21.5.4. Security and Fairness

Inference engineering also has security implications. Because tokens consume finite physical resources, input and output limits also function as protection mechanisms against resource exhaustion.

Several controls are particularly important:

- **Input token limits prevent arbitrarily large contexts:** Users should not be able to consume compute and KV cache without bounds;
- **Output limits control runaway generation:** Excessively long responses increase cost and keep capacity occupied;
- **Per-tenant quotas protect fairness:** A single customer should not be able to consume the entire GPU pool or KV cache;
- **Admission control protects SLOs:** Requests can be rejected or queued before the service enters severe saturation;
- **Model artifacts require integrity controls:** Manipulated weights compromise the entire serving layer;
- **Sensitive prompts require protection:** Inference logs may contain confidential information and should not be treated as ordinary telemetry.

Fairness, therefore, is not solely a property of the scheduler. It depends on quotas, limits, admission control, and resource isolation. A shared service must prevent the workload of one tenant from degrading the experience of others in an uncontrolled manner.

21.5.5. Economics and Major Trade-Offs

Inference engineering is also an economic discipline. Serving decisions directly affect cost per request, cost per token, and cost per successful task.

The primary trade-offs include:

- **Latency vs. Throughput:** Larger batches can improve utilization while increasing individual waiting time;
- **Context Length vs. Concurrency:** More tokens per sequence reduce the number of simultaneous users;
- **Model Size vs. Cost:** Larger models can improve quality but reduce density per GPU and increase infrastructure requirements;
- **Quantization vs. Quality:** Lower precision reduces memory footprint but may degrade behavior in specific segments;
- **More Replicas vs. Utilization:** Idle capacity reduces queueing but worsens economics;
- **Prefix Caching vs. Freshness:** Reuse reduces compute but requires correct invalidation;
- **Fallback Quality vs. Availability:** An alternative model keeps the service operational but may reduce semantic quality;
- **Streaming vs. Post-Validation:** Displaying tokens immediately improves UX but reduces the opportunity to validate the complete response before exposure.

None of these decisions should be evaluated solely in terms of local technical gain. A reduction in TTFT that requires twice as much hardware may be excellent for a product whose conversion depends on immediate response and economically infeasible in another context. Similarly, a larger model may justify its cost for a high-value task while representing waste for a low-impact feature.

The most important economic metric is not necessarily cost per token. It is the **cost of producing a successful task at sufficient quality**.

21.5.6. Important Anti-Patterns

Many production inference problems arise when convenient metrics are used as inadequate proxies for the real behavior of the system. Several anti-patterns deserve particular attention:

- **Measuring only requests per second hides radical differences in workload:** A request containing thousands of tokens is not equivalent to a short query;
- **Optimizing only GPU utilization confuses utilization with outcome:** 100% GPU utilization can coexist with high TTFT and poor unit economics;
- **Using enormous batches for interactive traffic maximizes local throughput:** However, it can significantly degrade the user experience;
- **Ignoring KV cache in capacity planning considers only weights:** The model may fit while the required concurrency does not;
- **Autoscaling only by GPU utilization reacts to an incomplete signal:** Queue and token pressure may grow before compute reaches saturation;
- **Using quantization without a regression suite introduces invisible risk:** Quality losses may appear in rare and critical tasks;
- **Applying round-robin across replicas with heterogeneous workloads creates imbalance:** Uniform request count does not imply uniform work;
- **Using fallback without semantic evaluation confuses availability with quality:** HTTP success can mask severe regression;

- **Repeatedly processing large system prompts without caching predictably wastes compute:** Stable prefixes should be treated as opportunities for reuse;
- **Generating excessive outputs by default prolongs decode:** Throughput and cost degrade without proportional benefit;
- **Selecting a model solely by benchmark ignores serving economics:** A model that is academically superior may be operationally infeasible.

These anti-patterns share one characteristic: **they optimize or observe an isolated component without considering end-to-end system behavior.** At Staff level, the responsibility is to connect workload, runtime, hardware, product, and economics into a single decision model.

Chapter Summary

LLM Inference Engineering transforms a trained model into an operational capability capable of serving real-world workloads at scale. The problem involves far more than loading weights onto a GPU. Performance emerges from the interaction among prefill, decode, memory bandwidth, KV cache, batching, scheduling, parallelism, replicas, network, and the characteristics of the requests themselves.

The core principles can be summarized as follows:

- **Separate prefill and decode:** The two phases have different bottlenecks and metrics;
- **Use TTFT and TPOT to diagnose latency:** Total latency alone hides the source of degradation;
- **Treat KV cache as a first-class resource:** In many workloads, it limits concurrency before compute does;
- **Plan capacity in tokens and active sequences:** Request count poorly represents actual work;
- **Use prefix caching to eliminate repeated work:** Large, frequently reused prefixes represent clear optimization opportunities;
- **Prefer continuous batching for heterogeneous workloads:** Requests can enter and leave dynamically without leaving slots idle;
- **Design scheduling around product requirements:** Fairness, priority, TTFT, and throughput must reflect real SLOs;
- **Use quantization when memory or bandwidth are limiting resources:** Gains must be validated against quality regressions;
- **Evaluate weight-only quantization end-to-end:** Smaller storage does not guarantee faster execution;
- **Use speculative decoding when acceptance rate justifies the overhead:** Drafting creates value only when it avoids enough target-model work;
- **Use model parallelism only to solve a concrete constraint:** Communication overhead must remain lower than the benefit obtained through distribution;
- **Keep communication-intensive ranks close together:** Topology matters once serving spans multiple devices or nodes;
- **Balance replicas based on estimated work:** Active tokens, queue depth, and KV cache pressure represent load better than request count;
- **Autoscale using relevant saturation signals:** Queue depth, active sequences, and KV cache can be more informative than generic metrics;
- **Account for startup time in autoscaling:** Warm capacity may be necessary when loading new replicas is slow;
- **Use model routing to improve quality per dollar:** Frontier capability does not need to serve every request;

- **Distinguish system latency from perceived latency:** Streaming can improve UX without changing completion time;
- **Design fallbacks for controlled semantic degradation:** Availability should not be measured only through HTTP status;
- **Observe throughput, latency, capacity, and cost jointly:** A local optimization can degrade the complete system;
- **Control tokens as both an economic and security resource:** Input limits, output limits, quotas, and admission control protect shared capacity;
- **Connect serving decisions to the product:** Context size, output length, and model tier are also business decisions.

The central lesson is that efficient LLM serving requires managing a dynamic system of sequences with different lengths, states, priorities, and costs. GPU compute is only one of the relevant resources. Scheduler behavior, KV cache, memory bandwidth, topology, and workload shaping can determine a substantial portion of observed performance.

For a Staff AI Engineer, this changes the fundamental question. Instead of searching for the technology that produces the highest isolated benchmark, **the goal is to identify the configuration that delivers sufficient quality within the product's latency, capacity, reliability, and cost constraints.**

Practical Framework for Designing LLM Inference

When analyzing an inference workload, the first step is to characterize the actual load. Architecture should not be designed around an abstract average request because token distributions, concurrency, and priorities determine a substantial portion of system behavior.

Start by understanding the workload:

- **Requests per second:** How many requests per second are received?;
- **Input distribution:** What is the distribution of input tokens?;
- **Output distribution:** What is the distribution of output tokens?;
- **Context percentiles:** What are the p50, p95, and p99 context lengths?;
- **Concurrency:** How many sequences need to remain active simultaneously?;
- **Workload type:** Is the workload interactive, batch, or mixed?;
- **Streaming:** Is streaming required?

After understanding the workload, translate product requirements into explicit SLOs. Without these boundaries, any inference optimization becomes difficult to evaluate.

Define the SLOs:

- **TTFT:** What Time to First Token is acceptable?;
- **TPOT:** What Time per Output Token is acceptable?;
- **Total latency:** What completion time can the product tolerate?;
- **Throughput:** What volume must be sustained?;
- **Saturation behavior:** What level of degradation is acceptable during saturation?;
- **Workload classes:** Do interactive and offline workloads require different SLOs?

Once the workload and SLOs are established, decompose the critical path. Begin with prefill because long contexts, queueing, and repeated work can dominate TTFT.

Analyze prefill:

- **Prefill contribution:** How much of TTFT comes from prefill itself?;
- **Queueing contribution:** How much of TTFT comes from queue time?;
- **Prompt size:** Are prompts larger than they actually need to be?;
- **Context redundancy:** Is redundant information present in the context?;
- **Shared prefixes:** Is a large portion of the prefix shared across requests?;
- **Prefix caching:** Would prefix reuse materially reduce compute?;
- **Context value:** Is long context actually improving task success?

This analysis helps distinguish situations where more hardware is necessary from situations where simply reducing work provides greater benefit. Next, analyze decode:

- **Current TPOT:** What generation speed is being observed?;
- **Output necessity:** Is the average output length actually necessary?;
- **Bandwidth:** Is decode memory-bandwidth-bound?;
- **Concurrency impact:** Is high concurrency degrading tokens per second?;
- **Streaming:** Is streaming reducing perceived latency?;
- **Speculation:** Is there a real opportunity for speculative decoding?

When decode represents the bottleneck, the problem may exist either in the runtime or in product behavior. Reducing unnecessary outputs can be as relevant as changing hardware. Next, analyze KV cache and memory capacity:

- **Memory headroom:** How much memory remains after loading weights?;
- **Per-sequence footprint:** How much KV cache does each active sequence consume?;
- **Worst-case capacity:** What maximum concurrency fits within the worst context bucket?;
- **Fragmentation:** Is there meaningful fragmentation?;
- **Memory allocation:** Does the runtime use paged allocation or efficient blocks?;
- **Long-running sequences:** Are some sequences consuming excessive capacity?;
- **Tenant limits:** Is it necessary to limit context or output per tenant?

This step determines whether the system is compute-bound or capacity-bound. This distinction is essential because adding compute does not solve insufficient memory for new sequences. Next, analyze batching:

- **Static batching:** Is it compatible with workload heterogeneity?;
- **Continuous batching:** Would it increase utilization and effective concurrency?;
- **Effective batch size:** What batch size actually occurs in production?;
- **Head-of-line blocking:** Are long requests blocking short requests?;
- **Mixed lengths:** Can the scheduler efficiently combine different sequence lengths?;
- **TTFT impact:** Is increased throughput degrading TTFT?

The objective is not to maximize batch size, but to identify a policy that uses hardware efficiently without violating the SLOs of different traffic classes. Next, evaluate scheduling:

- **Interactive priority:** Do human requests receive priority over offline jobs?;
- **Starvation:** Is there a risk that long sequences never receive sufficient progress?;
- **Tenant tiers:** Do different customers require different priorities?;
- **Queue contribution:** Does queue time already represent a meaningful portion of latency?;
- **Scheduler objective:** Is the scheduler maximizing throughput or the desired experience?;
- **Admission control:** Are requests controlled before complete saturation occurs?

This analysis explicitly connects scheduling to business priorities. Next, evaluate quantization:

- **Memory bottleneck:** Are weights the primary memory consumer?;
- **INT8:** Does the format provide sufficient benefit?;
- **INT4:** Is the quality loss acceptable for this workload?;
- **Kernel support:** Does the hardware provide efficient kernels for the chosen representation?;
- **Execution benefit:** Does quantization reduce TPOT or only storage?;
- **Regression coverage:** Is there a suite covering reasoning, tool use, and long-tail tasks?;
- **Quality economics:** Does the concurrency gain compensate for any quality loss?

Quantization should be introduced as a measurable hypothesis, not as an automatic optimization. Next, analyze speculative decoding:

- **Decode bottleneck:** Is decode truly the limiting factor?;
- **Draft model:** Is there a model fast enough to propose tokens?;
- **Acceptance rate:** What acceptance rate is observed?;
- **Draft overhead:** How much additional compute is introduced?;
- **Workload stability:** Does the acceleration persist across different task types?;
- **Resource impact:** Does speculation increase memory pressure or scheduler complexity?;
- **Operational value:** Does the net gain justify operating an additional model?

This step prevents additional complexity from being introduced into workloads where speculation itself provides little return. Next, evaluate model parallelism:

- **Single-GPU fit:** Does the model truly need to be distributed?;
- **Tensor Parallelism:** Is it required for memory or compute?;
- **Interconnect:** Do the GPUs have sufficiently fast links?;
- **Communication:** Does device-to-device transfer appear in the critical path?;
- **Pipeline Parallelism:** Does it reduce memory without excessively increasing latency?;
- **Alternative architecture:** Could a configuration with fewer GPUs and quantization produce an equivalent result?;
- **Parallelism degree:** What is the lowest degree of parallelism capable of satisfying the SLO?

The objective should be to use the least distributed configuration capable of satisfying the requirements because every additional boundary increases communication overhead and failure surface. Next, analyze replicas and load balancing:

- **Replica count:** How many replicas are required to support the expected peak?;
- **Round-robin:** Does it effectively distribute work?;
- **Token-aware balancing:** Does the load balancer consider active tokens or only requests?;
- **Queue imbalance:** Does any replica have a significantly larger queue?;
- **KV-aware routing:** Does KV cache pressure participate in the routing decision?;
- **Health management:** Is a degraded replica quickly removed from the pool?

Effective load balancing must distribute work, not simply request counts. Next, analyze autoscaling:

- **Saturation signal:** Which metric best represents capacity loss?;
- **Queue behavior:** Does queue depth increase before GPU utilization?;
- **Token signal:** Do active tokens better represent the workload?;
- **Startup time:** How long does a new replica take to load the model?;
- **Reactive scaling:** Is it fast enough for the traffic pattern?;

- **Warm capacity:** Is it necessary to maintain capacity ahead of demand?;
- **Predictive scaling:** Does traffic seasonality justify forecasting?;
- **Idle economics:** How much idle capacity is economically acceptable?

Autoscaling must be evaluated jointly with SLOs and the cost of headroom. Provisioning too late can be as expensive for the product as keeping GPUs idle. Next, evaluate model routing:

- **Model necessity:** Does every request truly require the current model?;
- **Lower tiers:** Which tasks could use smaller models?;
- **Under-routing cost:** What is the impact when a task receives insufficient capability?;
- **Over-routing cost:** How much is wasted by using models that are unnecessarily capable?;
- **Rule-based routing:** Would simple rules already be sufficient?;
- **Escalation:** Are confidence or a verifier required to decide when to move to a higher tier?;
- **Frontier traffic share:** What percentage of workload actually reaches the most expensive model?

Routing should be evaluated based on the combination of quality and economics, not solely cost reduction. Next, analyze reliability:

- **Replica failure:** What happens if a worker dies during generation?;
- **Retry economics:** Is rerunning the request safe and economically acceptable?;
- **Timeout:** What time limit exists per request?;
- **Fallback:** Is there an alternative model?;
- **Semantic degradation:** How much quality loss does the fallback introduce?;
- **Graceful degradation:** Can the system respond partially or disable some features?;
- **Regional requirements:** Is multi-region operation required?

This analysis should incorporate failures into the original design rather than addressing them only after incidents occur. Next, evaluate observability:

- **Latency decomposition:** Are TTFT, TPOT, and queue time measured separately?;
- **Token distributions:** Are input and output tokens monitored as distributions?;
- **KV cache:** Is its utilization visible?;
- **Concurrency:** Are active sequences monitored?;
- **Batching:** Is effective batch size known?;
- **GPU efficiency:** What is the tokens-per-second rate per GPU?;
- **Context segmentation:** Which context buckets exhibit the worst latency?;
- **Cost attribution:** Can cost be attributed by tenant, feature, and model tier?

If these questions cannot be answered through telemetry, the serving layer does not yet have sufficient observability for troubleshooting at scale. Next, evaluate security and fairness:

- **Input limits:** Is there a limit on input tokens?;
- **Output limits:** Is there a generation limit?;
- **Tenant quotas:** Does each tenant have controlled capacity?;
- **Noisy neighbors:** Can a single user consume all available concurrency?;
- **Admission control:** Is full saturation prevented before collapse?;
- **Sensitive logging:** Are confidential prompts stored in logs?;
- **Artifact protection:** Do model weights and artifacts have integrity and access controls?

These controls make resource management a security layer as well. Next, analyze economics:

- **Token economics:** What is the cost per million tokens served?;
- **Outcome economics:** What is the cost per successful task?;
- **Idle capacity:** How much GPU idle time exists?;
- **KV density:** How much does KV cache pressure reduce density?;
- **Quantization economics:** Would lower precision allow the number of GPUs to be reduced?;
- **Caching economics:** Does prefix caching materially reduce compute?;
- **Smaller models:** Could a smaller model serve a large portion of traffic?;
- **Output economics:** Would shorter outputs reduce cost without reducing value?;
- **Self-hosting economics:** Does self-hosting remain cheaper once engineering, reliability, and on-call responsibilities are included in the calculation?

This step prevents infrastructure cost from being analyzed independently of the engineering cost required to operate the system. Finally, connect inference to business outcomes:

- **Latency value:** What level of latency actually affects conversion, adoption, or user satisfaction?;
- **Quality delta:** How much quality is lost when using a smaller model?;
- **Economic value of quality:** How much is that incremental quality worth to the business?;
- **Peak capacity:** What is the cost of keeping resources available for spikes?;
- **Future growth:** How much additional throughput will be required?;
- **Platform leverage:** Can the architecture support new products without duplicating the entire serving stack?;
- **Optimization target:** Are we optimizing GPU benchmarks or the product's final KPI?;
- **Complexity budget:** If the serving architecture were simplified by half, what real impact would occur in quality, latency, and cost?

This is the mindset expected of a Staff AI Engineer.

LLM serving should not be treated as the simple deployment of a model. It is a system involving scheduling, memory, compute, network, and economics in which workload shaping, KV cache, batching, routing, capacity planning, and reliability must be designed jointly.

The correct architecture is not necessarily the one that maximizes tokens per second, uses the most expensive GPU, or implements the largest number of optimizations: **it is the architecture that delivers sufficient quality, within the required SLOs, with adequate reliability and an economically sustainable cost for the product.**