Inference and Scaling: Optimizing a Cloud AI Platform's LLM Serving Infrastructure

Stratos AI -- Reducing Latency and Cost for Multi-Tenant LLM Inference at Scale

Section 1: Industry Context and Business Problem

The Company

Stratos AI is a Series B cloud AI platform startup (\$47M raised) that provides managed LLM inference as a service to enterprise customers. Founded in 2022 by former Google Cloud and NVIDIA engineers, the company runs a multi-tenant GPU cluster serving language model inference for over 120 enterprise clients across fintech, legal tech, and customer support automation.

Stratos operates a fleet of 256 NVIDIA A100 80GB GPUs across two data centers (us-east-1 and eu-west-1). Their core product, **StratosServe**, allows enterprises to deploy fine-tuned language models (7B to 70B parameters) behind a unified API, handling tokenization, inference, and response streaming. Clients pay per million tokens generated, with pricing tiers based on model size and latency SLA.

The Problem

Stratos is hitting a wall. As their customer base has grown from 40 to 120 clients over the past 9 months, three compounding problems have emerged:

1. KV cache memory pressure. Each concurrent request allocates KV cache memory proportional to the sequence length. For their flagship 13B parameter model (40 attention layers, hidden dimension 5120, 40 heads), a single request with a 4096-token context requires:

$$\text{KV cache per request} = 2 \times 40 \times 4096 \times 5120 \times 2 \text{ bytes} = 3.36 \text{ GB}$$

With 80 GB per GPU, this limits concurrency to roughly 20 simultaneous requests per GPU after accounting for the model weights (26 GB in fp16). During peak hours (9 AM -- 5 PM EST), the platform receives 180+ concurrent requests per GPU pool, forcing aggressive request queuing that inflates tail latency.

2. Latency SLA violations. Stratos guarantees a p99 time-to-first-token (TTFT) of 500 ms and a p99 inter-token latency of 50 ms for their premium tier. Over the past quarter, the platform has violated these SLAs in 12% of peak-hour windows. Each SLA violation triggers a credit to the affected customer -- costing Stratos an average of \$38,000 per month in credits.

3. Inference cost inefficiency. Current GPU utilization during generation (the decode phase) averages only 15-22% of peak FLOPS. The decode phase is memory-bandwidth-bound because each step processes a single token and performs a single matrix-vector multiplication per layer, vastly underutilizing the GPU's compute capacity. This means Stratos is paying for compute it cannot use, directly eroding margins.

Business Impact

The financial pressure is severe:

- **Revenue:** \$4.2M monthly recurring revenue (MRR) across 120 clients.
- **GPU infrastructure cost:** \$1.8M/month (43% of revenue). Healthy margin requires this below 30%.
- **SLA credits:** \$38K/month and growing as client base expands.
- **Churn risk:** 8 enterprise clients (representing \$640K ARR) have flagged latency as a potential reason for non-renewal.
- **Competitive pressure:** Three rival platforms have launched with lower latency claims. Stratos must improve inference throughput by at least 3x without proportional cost increase, or face margin collapse.

Constraints

Constraint	Value	Rationale
GPU fleet	256x A100 80GB	No new hardware procurement for 6 months (supply chain)
Model sizes	7B, 13B, 70B parameters	Must support all three tiers
Latency SLA	TTFT < 500 ms, inter-token < 50 ms (p99)	Contractual obligation
Accuracy	Sampling output must be statistically identical to baseline	Cannot degrade model quality
Deployment timeline	8 weeks to production	Board-mandated deadline
Team	6 inference engineers, 2 systems engineers	No hiring budget this quarter

Section 2: Technical Problem Formulation

Problem Decomposition

Stratos's inference pipeline has three distinct bottlenecks, each mapping directly to a concept from the Inference and Scaling article:

Bottleneck 1: KV Cache Memory Management

The naive approach allocates a fixed KV cache buffer for the maximum possible sequence length (4096 tokens) at request arrival, even if most requests use far fewer tokens. Internal telemetry shows the median generated sequence length is only 847 tokens, meaning 79% of allocated cache memory is wasted on average.

The memory waste per GPU can be quantified:

$$\text{Waste ratio} = 1 - \frac{\text{median actual length}}{\text{max allocated length}} = 1 - \frac{847}{4096} = 0.793$$

At 20 concurrent requests, this wastes approximately $20 \times 3.36\ \text{GB} \times 0.793 = 53.3\ \text{GB}$ of the 80 GB available -- memory that could serve 15 additional requests.

Bottleneck 2: Sampling Overhead

Stratos currently applies temperature scaling and top-p sampling on the CPU after transferring logits from GPU. For a vocabulary of 32,000 tokens, this transfer and CPU-side sort operation adds 2-8 ms per token. Over a 200-token generation, this accumulates to 400-1600 ms of unnecessary latency.

The sampling operation itself involves sorting the full vocabulary:

$$\text{Sort cost} = O(|V| \log |V|) = O(32000 \times 15) = O(480{,}000) \text{ operations pe}$$

Bottleneck 3: Decode-Phase Underutilization

During the decode phase, each forward pass processes a single token. The dominant operation is the KV projection, which is a matrix-vector multiplication:

$$\mathbf{q} = W_Q \mathbf{x}, \quad \mathbf{k} = W_K \mathbf{x}, \quad \mathbf{v} = W_V \mathbf{x}$$

where $W_Q, W_K, W_V \in \mathbb{R}^{d \times d}$ and $\mathbf{x} \in \mathbb{R}^d$. A matrix-vector multiply has arithmetic intensity (FLOPS per byte of memory accessed) of approximately:

$$\text{Arithmetic intensity} = \frac{2d^2}{(d^2+d) \times 2} \approx \frac{2d}{2} = d$$

For $d = 5120$, this is roughly 5120 FLOPS per 2 bytes, or about 2560 FLOPS/byte. But the A100's compute-to-bandwidth ratio is approximately 312 TFLOPS / 2 TB/s = 156 FLOPS/byte. Since our arithmetic intensity (2560) exceeds the hardware ratio (156), this single-token decode is actually compute-bound in theory -- but in practice, kernel launch overhead, memory allocation latency, and small batch sizes prevent the GPU from reaching anywhere near peak throughput.

The solution: **batch multiple requests together** during the decode phase (continuous batching), so that the matrix-vector multiply becomes a matrix-matrix multiply, dramatically improving GPU utilization.

Optimization Strategy

The engineering plan targets each bottleneck:

1. **Paged KV cache** (inspired by PagedAttention / vLLM): Allocate KV cache in fixed-size pages rather than contiguous blocks, enabling dynamic memory allocation and deallocation as sequences grow and complete.

2. **GPU-side fused sampling kernel**: Move temperature scaling, top-k filtering, top-p filtering, and multinomial sampling into a single fused CUDA kernel that operates entirely on-GPU, eliminating the CPU round-trip.

3. **Continuous batching with iteration-level scheduling**: Instead of waiting for an entire batch to complete, insert new requests into the batch at every decode iteration, maximizing GPU utilization.

Success Metrics

Metric	Current	Target
Concurrent requests per GPU (13B model)	20	60+
GPU memory utilization efficiency	21%	75%+
p99 TTFT (13B model)	720 ms	< 400 ms
p99 inter-token latency	62 ms	< 35 ms
Decode-phase GPU FLOPS utilization	18%	55%+
Throughput (tokens/second/GPU)	340	1200+

Section 3: Implementation Notebook Structure

The implementation notebook walks through building each optimization component from scratch, measuring the impact at every step.

3.1 Baseline Inference Engine

Build a minimal transformer inference engine and establish baseline performance measurements.

TODO 1: Implement naive autoregressive generation without KV cache. Build a forward pass that reprocesses the entire sequence at every step. Measure tokens-per-second and memory usage as a function of sequence length. This establishes the worst-case baseline.

TODO 2: Add a standard KV cache. Implement the basic KV cache optimization from the article. Store K and V tensors per layer, append at each step, and feed only the new token through the model. Measure the speedup and compare memory usage against the naive approach.

3.2 Paged KV Cache

Implement a paged memory manager that allocates KV cache in fixed-size blocks.

TODO 3: Build a page table and block allocator. Implement a PagedKVCache class that maintains a pool of fixed-size memory blocks (e.g., 16 tokens per block) and a page table mapping (request_id, layer, position) to block indices. Implement allocate, append, and free operations.

TODO 4: Measure memory efficiency. Run 50 concurrent simulated requests with variable output lengths drawn from Stratos's observed distribution (log-normal, median 847 tokens). Compare total memory consumption between the standard contiguous KV cache and the paged KV cache. Compute the effective concurrency improvement.

3.3 GPU-Optimized Sampling

Move sampling from CPU to GPU and fuse the operations.

TODO 5: Implement fused temperature + top-p sampling on GPU. Write a sampling function that performs temperature division, softmax, cumulative sum, masking, and multinomial sampling entirely in PyTorch on the GPU -- no CPU transfers. Benchmark against the CPU baseline and measure the per-token latency reduction.

TODO 6: Compare sampling strategies. For a fixed prompt, generate 100 completions under greedy, temperature-only ($\tau = 0.7$), top-k ($k = 40$), and top-p ($p = 0.9$) sampling. Measure output diversity (distinct n-grams), latency per token, and qualitative coherence. Visualize the probability distribution under each strategy.

3.4 Continuous Batching

Implement iteration-level scheduling to maximize GPU utilization.

TODO 7: Build a continuous batching scheduler. Implement a scheduler that maintains a running batch of active requests. At each decode iteration, the scheduler: (a) removes completed requests (those that have generated an EOS token or reached max length), (b) inserts waiting requests from the queue into freed slots, and (c) pads the KV cache appropriately for the new heterogeneous batch.

TODO 8: Measure throughput under load. Simulate a Poisson arrival process with $\lambda = 10$ requests/second, each requesting 100-500 generated tokens. Compare throughput (total tokens generated per second) and latency (p50, p95, p99 TTFT and inter-token latency) between static batching and continuous batching.

3.5 LoRA Serving: Multi-Tenant Model Adaptation

Serve multiple LoRA-adapted models from a single base model.

TODO 9: Implement LoRA adapter loading and switching. Build a LoRAManager that stores multiple LoRA adapters (A, B matrices for each target layer) in GPU memory. Given a request tagged with a client ID, the manager applies the correct adapter during the forward pass. Demonstrate serving 3 different fine-tuned variants from one base model with negligible memory overhead.

TODO 10: End-to-end system benchmark. Combine all optimizations (paged KV cache, GPU sampling, continuous batching, multi-tenant LoRA) into a single inference pipeline. Run the Poisson load test again and compare against the original baseline. Report the final throughput, latency, and memory utilization numbers.

Section 4: Production and System Design Extension

Scaling Law Application: Right-Sizing the Model Fleet

Stratos serves three model sizes: 7B, 13B, and 70B. The Chinchilla scaling law provides a framework for advising clients on which model tier to choose.

For a client considering the 70B model, the engineering team computes the expected quality improvement over the 13B model using the Kaplan scaling exponent $\alpha_N \approx 0.076$:

$$\frac{L(70B)}{L(13B)} = \left(\frac{13B}{70B}\right)^{0.076} = (0.186)^{0.076} = 10^{0.076\times\log_{10}(0.186)} = 10^{0.076\times(-0.73}$$

A 12% loss reduction. But the 70B model costs 5.4x more per token to serve (proportional to parameter count, adjusted for lower decode efficiency at larger sizes). For many enterprise use cases -- customer support, document summarization, code completion -- the quality gap does not justify the cost. Stratos uses this analysis to route 60% of their traffic to the 13B tier, substantially improving overall fleet economics.

Production Architecture

The production system consists of:

1. **Request Router.** An NGINX-based load balancer that routes requests to the appropriate model pool based on client SLA tier and model size. It implements token-bucket rate limiting per client and priority queuing for premium tier.

2. **Inference Engine Pool.** Each GPU runs a single inference engine process that implements paged KV cache, GPU-side sampling, and continuous batching. The engine exposes a gRPC interface for low-latency internal communication.

3. **LoRA Adapter Registry.** A centralized service backed by Redis that stores adapter metadata (client ID, model size, adapter weights S3 path, version hash). When a request arrives for a client with a custom fine-tuned model, the engine loads the corresponding LoRA adapter on first access and caches it in GPU memory using an LRU eviction policy.

4. **Monitoring and Autoscaling.** Prometheus metrics track per-GPU KV cache utilization, request queue depth, SLA compliance rate, and decode throughput. A custom Kubernetes HPA (Horizontal Pod Autoscaler) scales GPU pods based on queue depth and SLA violation rate, not just CPU/memory utilization.

Failure Modes and Mitigations

Failure Mode	Impact	Mitigation
KV cache OOM during generation	Request killed mid-stream	Pre-flight memory check: reject requests that would exceed 90% cache capacity; gracefully return partial response with continuation token
LoRA adapter version mismatch	Silent quality degradation	Adapter version hash verified at load time; stale adapters trigger re-download
GPU hardware failure	Pool capacity reduced	N+2 redundancy per pool; automatic failover in < 30 seconds
Adversarial long-context inputs	Memory exhaustion attack	Hard 4096 token input limit enforced at router; per-client concurrency caps
Continuous batching starvation	Low-priority requests starved	Aging-based priority boost: requests waiting > 2 seconds get promoted to high priority

Cost Analysis

After deploying all optimizations, Stratos projects the following cost impact:

Metric	Before	After	Improvement
Tokens/second/GPU (13B)	340	1,200+	3.5x
Concurrent requests/GPU	20	60+	3x
GPU cost per million tokens	0.84 USD	0.24 USD	3.5x reduction
Monthly GPU infrastructure cost	1.8M USD	0.78M USD	1.02M USD savings
SLA violation rate (peak hours)	12%	< 1%	Contractual compliance restored
SLA credit payouts	38K USD/ month	< 3K USD/ month	35K USD/month savings

Metric	Before	After	Improvement
Gross margin on inference	57%	81%	Sustainable unit economics

The combined savings of 1.06 million USD per month (12.7 million USD annualized) exceeds the 47 million USD total capital raised, demonstrating that inference optimization is not a nice-to-have -- it is the difference between a viable business and one that burns through runway serving tokens at a loss.