vmware by Broadcom

SPOC

A Stateful, Profile-Based Optimization for LLM Capacity Planning Methodology

White Paper

Table of Contents

Chapter 1: Abstract

Deploying a large language model (LLM) in production requires knowing in advance how many concurrent users the system can serve while meeting strict latency commitments. Standard benchmarking approaches, which often rely on fixed-length, stateless synthetic prompts at uniform concurrency, fail to reflect the heterogeneous, multi-turn nature of real developer-assistant traffic. Consequently, they cannot reliably predict where a production system will fail, particularly when handling massive, accumulating context windows.

This paper introduces stateful, profile-based optimization for LLM capacity (SPOC), a benchmarking framework built to answer a specific question, “How many concurrent AI coding-assistant users can a given hardware configuration sustain while keeping the 95th-percentile time-to-first-token (TTFT) below 10 seconds and the 95th-percentile inter-token latency (ITL) below 200 milliseconds?”

Crucially, this document is designed as a reproducible methodology and practical guide. Rather than just presenting our final numbers, we explain the fundamental concepts of performance monitoring, stateful stress testing, and multi-objective optimization. The goal is to equip infrastructure engineers with the theoretical grounding and practical toolset required to adapt the SPOC framework to their own specific models, hardware, and user populations.

To achieve accurate capacity planning, SPOC simulates reality. Rather than sending isolated queries, the framework maintains true statefulness, simulating continuous, multi-turn dialogues where context progressively accumulates. We define three distinct user profiles based on observations over one internal development team's behavior: P0 (tactical developer, 70%), P1 (analytical developer, 20%), and P2 (forensic analyst, 10%), pushing the context boundary up to 128,000 tokens. Furthermore, because real developers often query a shared foundational code base while working on unique local branches, we introduce a partial common ground dataset geometry. This design accurately simulates enterprise memory pressure by balancing realistic prefix-cache utilization against heavy, unique prefill compute.

To navigate the complex configuration space of modern inference engines under this realistic load, SPOC employs a multi-stage approach. First, we use a Locust load generator to apply a stepped optimization strategy, incrementally increasing user concurrency to find the absolute capacity ceiling. Second, we conduct an exhaustive multi-objective search using Optuna's implementation of the NSGA-II evolutionary algorithm. This search identifies the Pareto-optimal configurations, the best possible trade-offs between throughput and latency, for high-concurrency serving.

We validate the SPOC methodology on two distinct hardware configurations (4x NVIDIA H100 and 2x NVIDIA H200) running the `gpt-oss-120b` model via vLLM v0.16.0. Finally, we provide deep kernel-level profiling using NVIDIA Nsight Systems to explain the fundamental hardware bottlenecks at scale.

Chapter 2: Introduction

2.1 The Capacity Planning Problem

LLM inference capacity planning is fundamentally a resource allocation problem under uncertainty. The abstract question, "How many users can this server handle?" conceals a more precise one: “How many users of this specific kind, doing this specific mix of tasks, can be served while keeping latency within these specific bounds?”

The answer is highly sensitive to all three variables. A server that comfortably handles 300 users sending short documentation queries may fail at 100 users if those users are submitting 80,000-token code bases for analysis. A latency threshold of 10 seconds for the first token is appropriate for an interactive assistant handling massive contexts; a batch processing pipeline might tolerate 60 seconds. A workload where 70% of requests are short and 10% are very long stresses the system differently than one where all requests are medium-length, even if the average prompt size is identical.

This sensitivity means that capacity cannot be read off a hardware spec sheet or inferred from a single-prompt benchmark. It must be measured under a workload that accurately represents the target user population, with latency measured in the way users actually experience it. For streaming LLM responses, that means separating TTFT, which is the wait before the response begins, from ITL, which measures the smoothness of the stream once it starts. These two metrics respond to different aspects of system configuration and must be optimized jointly.

The standard approach of running a benchmark at a fixed concurrency level with a representative prompt, fails on both counts. Most benchmark tools generate requests from a uniform or Poisson distribution, which does not capture the heterogeneous nature of real traffic. Furthermore, stateless benchmarks fail to simulate the progressive accumulation of KV cache that occurs in a real multi-turn AI assistant session.

2.2 What This Work Contributes

This paper describes a framework for answering the capacity planning question rigorously, updated for stateful, massive-context workloads. It was built and validated for the `gpt-oss-120b` model served by vLLM v0.16.0.

Throughout this white paper, each measurement stage is tied to operational choices, such as what to hold fixed, what to log, and when to stop a sweep, so the text can double as a runbook, not only as an experiment report.

The choice of `gpt-oss-120b` is deliberate. As a Mixture-of-Experts (MoE) architecture, it remains a highly competitive open-source language model that balances advanced reasoning capabilities with exceptional throughput and speed. Its sparse activation pattern allows it to serve high-concurrency workloads more efficiently than dense models of comparable parameter counts.

Similarly, vLLM v0.16.0 was selected because, at the time of writing, it proved to be the most stable release for NVIDIA H100 and H200 deployments. Subsequent releases (v0.17.0, v0.18.0, and v0.19.0) introduced documented regressions affecting performance and hardware compatibility for this specific MoE model on Hopper architectures; for more information, refer to community reports on MoE memory allocation regressions and CUDA graph capture instability on H100s. Consequently, v0.16.0 remains the recommended baseline for this benchmarking framework.

By combining a stateful profile-based workload model (driven by Locust), a stepped optimization strategy governed by a monotonic failure rule (Section 5.2, Capacity Discovery and Parameter Optimization), a Non-dominated Sorting Genetic Algorithm II (NSGA-II) multi-objective optimization loop (driven by Optuna), and a triple telemetry stack (vLLM Prometheus, DCGM Exporter, and NVIDIA Nsight Systems), this framework identifies the true Pareto-optimal configurations for high-concurrency serving and provides empirical evidence for the underlying hardware and software bottlenecks.

Chapter 3: System Under Test

3.1 Hardware and Virtualization Platform

All experiments were conducted on Linux virtual machines hosted on VMware Cloud Foundation® (VCF). The GPUs are assigned to the VM using VMDirectPath I/O (DirectPath I/O), the VMware mechanism for direct device assignment that bypasses the ESXi hypervisor and gives the guest OS exclusive, unmediated access to the physical hardware. For GPU-bound workloads, DirectPath I/O delivers the performance equivalent to bare-metal applications.

Configuration 1: 4x NVIDIA H100 (80 GB): The server operates on a VCF powered by a dual-socket Intel Xeon Platinum 8470 processor (104 physical cores total). The VM is configured to consume the host's entire CPU footprint (104 vCPUs, 1 thread per core) but only half of its GPU capacity (4x H100 out of 8 available in the physical host).

- Compute and memory: The VM is provisioned with 104 vCPUs and 1.0 TiB of RAM, consuming the entire physical host's compute footprint to prevent noisy-neighbor contention.
- GPU attachment: The 4 GPUs are attached via VMDirectPath I/O (PCIe passthrough), exposed directly on the guest's PCI bus.
- NUMA topology: The VM spans two NUMA nodes (Node 0 and Node 1). All four H100 GPUs are localized to NUMA Node 0. The first 52 vCPUs (cores 0 to 51) are affine to NUMA Node 0, ensuring that the inference engine can be pinned to local cores to avoid cross-node QPI/UPI latency penalties. In this study, a strict boundary constraint is enforced: a single tensor parallelism (TP) group MUST NEVER span across multiple NUMA nodes.
- GPU and interconnect: The underlying physical host is an HGX server utilizing 8x NVIDIA H100 SXM GPUs connected via NVSwitches. However, this VM is assigned a dedicated 4-GPU partition. Within this VM partition, the 4 GPUs maintain a full NV18 NVLink topology, meaning every assigned GPU has a direct, non-blocking link to the other three via NVSwitches. The theoretical bidirectional NVLink bandwidth per GPU remains 900 GB/s, providing the VM's 4-GPU partition with a total bisection bandwidth of 1.8 TB/s. Each GPU exposes approximately 80 GiB of usable HBM3 memory. The NVIDIA Driver 580.126.09 is installed in the VM. The inference engine runs within the official vLLM v0.16.0 Docker container pulled from DockerHub, which bundles its own CUDA 12.9 environment.

Configuration 2: 2x NVIDIA H200 (141 GB): The system is powered by an AMD EPYC 9555 64-Core Processor (Zen 4). The hypervisor presents 32 sockets with 1 core per socket and 1 thread per core to the guest OS.

- Compute and memory: The VM is provisioned with 2x H200 GPU (out of 4 available in the physical host), 32 vCPUs and 1.0 TiB (1007 GiB) of RAM.
- GPU attachment: The GPUs are attached via VMDirectPath I/O (PCIe passthrough), verified by the raw NVIDIA 3D controllers exposed directly on the guest's PCI bus.
- NUMA topology: Both H200 GPUs are localized to a single NUMA node (Node 0). All 32 vCPUs are affine to NUMA Node 0, ensuring zero cross-node QPI/UPI latency penalty, as all compute and memory access for the GPUs is strictly local. A strict boundary constraint is enforced: a single TP group MUST NEVER span across multiple NUMA nodes.
- GPU and interconnect: The system utilizes 2x NVIDIA H200-141C PCIe GPUs, each exposing exactly 141 GiB of usable HBM3e memory. They are connected via a direct NV6 topology (6 active NVLink connections), bridged using 3 physical NVLink bridges. The theoretical bidirectional NVLink bandwidth between these two GPUs is 600 GB/s. The NVIDIA Driver 580.82.07 is installed in the VM. The inference engine runs within the official vLLM v0.16.0 Docker container pulled from DockerHub, which bundles its own CUDA 12.9 environment.

3.2 Model Geometry and Arithmetic Intensity

Understanding the hardware's compute characteristics requires knowing exactly how `gpt-oss-120b` is represented at runtime. The model has 36 transformer layers with alternating sliding-window and full-attention patterns, a hidden size of 2880, 128 experts per layer in a MoE architecture with top-4 activation (4 of 128 experts selected per token), grouped-query attention, and a vocabulary of 201,088 tokens.

The model's weights are stored on disk in two precisions: the MLP expert projection weights are stored in MXFP4 (4-bit), while attention weights, router weights, embeddings, and layer norms are stored in BF16. At runtime on H100/H200, vLLM selects the `SM90_FI_MXFP4_BF16` back end. This is a W4A16 scheme: the MXFP4 expert weights are dequantized to BF16 in registers before each matrix multiplication, and all activations remain in BF16 throughout.

To understand how the system behaves as concurrency increases, it is useful to consider the theoretical roofline model. For example, the NVIDIA H100's BF16 ridge point is 299 ops per byte (1000 TFLOPS peak compute ÷ 3.35 TB/s HBM3 bandwidth). A traditional dense-model roofline analysis would compute arithmetic intensity [(2 × B × params_per_gpu) ÷ weight_bytes_hbm] and predict a clean crossover from memory-bound to compute-bound as the decode batch size (B) grows.

However, `gpt-oss-120b` is a sparse MoE model. Each token activates only 3.1% of the expert parameters (4 out of 128). At small batch sizes, this is highly efficient. But at large batch sizes (B >100), the probability distribution shifts: across the entire batch, most or all of the 128 expert weight shards are required by at least one token. This creates a severe mismatch: the memory traffic becomes dense (the GPU must load nearly all expert weights from HBM3), but the compute remains sparse (each token only multiplies against 3.1% of those weights). This dynamic breaks the simple roofline formula, making theoretical predictions unreliable and necessitating the empirical kernel-level profiling presented in Chapter 7 and Chapter 8.

3.3 Inference Engine Mechanics

The model is served by vLLM v0.16.0 using the V1 engine with tensor parallelism. The following vLLM features are active in all experiments:

- Chunked prefill: Splits long prompts into fixed-size chunks processed across multiple forward passes. Because vLLM co-schedules prefill and decode tokens in the same forward pass, an unchunked large prompt (such as a P2 request with 80,000 input tokens) would monopolize the entire forward pass with prefill work, leaving no capacity for decode tokens from other users. This delays those users' next token and spikes their ITL. Chunking bounds the prefill contribution per step to `MAX_NUM_BATCHED_TOKENS`, keeping decode latency predictable.
- `MAX_NUM_BATCHED_TOKENS`: A critical vLLM parameter that defines the absolute maximum number of tokens (both prefill and decode combined) the engine will process in a single forward pass (GPU step). By tuning this value, we control the maximum duration of a single step, directly trading off overall throughput against strict ITL guarantees.
- Automatic prefix caching: `--enable-prefix-caching` caches the KV blocks of shared system prompts and multi-turn conversation histories in GPU memory. This allows subsequent turns in a stateful session to bypass the compute-heavy prefill phase for previously processed tokens, drastically reducing TTFT for active users.
- Async scheduling: Overlaps the scheduler's decision-making with the GPU's execution of the previous forward pass. Without async scheduling, the GPU sits completely idle while this scheduling decision is being made. At high concurrency, this CPU pause grows and adds directly to ITL.
- FP8 KV cache: Stores key-value tensors in 8-bit floating point rather than BF16, halving the KV cache memory footprint and allowing approximately twice as many concurrent sequences to reside in cache.
- Multiple API servers: `--api-server-count 4` runs four OpenAI-compatible HTTP server processes behind a single port, distributing request parsing and eliminating head-of-line blocking where one slow request being serialized would delay all subsequent requests.

3.4 Monitoring Stack

The monitoring stack runs alongside the vLLM inference server:

- vLLM Prometheus endpoint: Exposes engine-internal counters and histograms, including the following: `vllm:time_to_first_token_seconds`, `vllm:inter_token_latency_seconds`, `vllm:num_requests_running`, `vllm:kv_cache_usage_perc`, and `vllm:prefix_cache_hit_rate`.
- DCGM Exporter: Exposes hardware-level GPU counters, including the following: `DCGM_FI_PROF_SM_ACTIVE`, `DCGM_FI_PROF_DRAM_ACTIVE`, and `DCGM_FI_DEV_NVLINK_BANDWIDTH_TOTAL`.
- NVIDIA Nsight Systems (NSYS): Used for deep kernel-level profiling at the capacity ceiling to decompose GPU active time into specific CUDA kernels (Flash Attention, MoE Expert GEMM, NCCL AllReduce). Methodologically, it is critical to directly wrap the vLLM process within the main container (`nsys profile ... vllm serve`) rather than using a sidecar container with `--attach`. The sidecar attach method is incompatible with the deep CUDA API tracing required for this level of performance analysis.

Chapter 4: Workload Design

4.1 Benchmarking Framework Methodology

To rigorously determine the maximum concurrency and optimal configuration for this environment, our methodology follows a structured framework:

1. Workload characterization and baseline: Establish an initial baseline configuration grounded in the physics of the GPUs and the MoE model's geometry.
2. Capacity discovery and headroom allocation: Search for the concurrency wall, but select a target operational concurrency, such as 300 users, that leaves a safe margin and headroom to absorb unpredictable traffic spikes that deviate from the standard P0/P1/P2 distribution.
3. Parameter optimization: Run the Optuna NSGA-II optimization at that target concurrency to find the best vLLM parameters that strictly satisfy SLAs.
4. The golden run and telemetry analysis: Run a 60-minute golden run (soak test) to prove stability over a sustained period, capturing deep hardware telemetry to prove why the system remains stable.

4.2 The Importance of True Statefulness

A critical advancement in this framework is the transition from stateless simulation to true statefulness. Traditional benchmarks often rely on stateless clients sending independent, static prompts. To simulate memory pressure, they might send massive payloads, but this fails to capture the temporal dynamics of a real AI assistant session.

In a true stateful workload, the load generator maintains active session histories. As a simulated developer engages in a multi-turn conversation, new user messages and model responses are progressively appended to the context window. This turn-by-turn accumulation stresses the inference engine's memory allocator (such as vLLM's PagedAttention) and scheduling pipeline in ways that static prompts cannot. It accurately reflects the memory fragmentation, dynamic KV cache growth, and eventual eviction pressures of a live enterprise deployment, ensuring that the measured capacity ceiling holds true under real-world conditions.

4.3 The Power-Law of Code Access and User Profiles

A realistic benchmark must reflect the actual distribution of traffic. In modern enterprise software development, code access follows a power-law distribution, often colloquially related to the Pareto principle or 80/20 rule. In this context, it means a small number of core repositories, foundational frameworks, and standard libraries are referenced by the vast majority of developers, while a massive long-tail of specific feature files are accessed only by the individual developers actively modifying them. When an AI coding assistant builds a prompt via Retrieval-Augmented Generation (RAG), it typically fetches this shared base and appends the unique work branch, meaning the specific local files the developer is editing.

To simulate this, our `partial-common-ground` dataset pregenerates a fixed pool of shared monorepos (60,000 tokens each) and shared core libraries (10,000 tokens). To populate the unique, conversational portions of the workload, we sample multi-turn developer trajectories from two open-source Hugging Face datasets: togethercomputer/CoderForge-Preview and nebius/SWE-rebench-openhands-trajectories. These datasets were selected because they contain authentic, complex

software engineering interactions, ranging from isolated function debugging to repository-wide feature implementations, rather than synthetic or single-turn queries. By extracting the raw user turns from these trajectories and combining them with our shared code pools, we construct a stateful dataset designed around three distinct user profiles, mirroring realistic enterprise AI assistant usage patterns:

- P0—Tactical Developer, 70%: Short-context, highly localized queries, such as general coding questions or debugging a single function. P0 users do not utilize the shared code pools; their prompts consist entirely of unique, localized context representing the long tail of isolated tasks. Sessions span 5 to 15 turns. As illustrated in Figure 1, these typically start with an initial payload of roughly 5000 unique tokens and can grow up to 50,000 tokens as the conversation history accumulates. Think-time of 30 to 90 seconds simulates a developer who reads the response before sending the next request.
- P1—Analytical Developer, 20%: Sessions span 16 to 40 turns. P1 users are randomly assigned one of the shared core libraries (10,000 tokens), augmented with unique local context. Figure 1 shows this resulting in a starting payload of 15,000 tokens (10,000 shared + 5000 unique), which grows up to 120,000 tokens over the session. Think-time of 120 seconds to 300 seconds reflects a slower read-and-respond cadence appropriate for more complex material.
- P2—Forensic Analyst, 10%: Sessions span more than 20 turns, loaded with massive contextual payloads to push the 128,000 context boundary. P2 users are randomly assigned one of several massive shared monorepos (60,000 tokens each), augmented with variable-length unique local context representing the developer's extensive uncommitted changes and task-specific modifications. This creates the ~75,000-token initial payload shown in Figure 1, representing a senior engineer submitting an entire repository code base alongside deep root-cause analysis queries. This deliberate sizing leaves ~50,000 tokens of headroom in the 128,000-token context window for the stateful accumulation of the conversation over 20+ turns. Think-time of 300 seconds to 900 seconds reflects the time needed to review a multi-thousand-token response.

NOTE:

- Note on think-times: These exceptionally long wait times (up to 15 minutes for P2) are not merely to simulate slow human reading speed. They are a deliberate benchmarking mechanism designed to force the inference engine to hold massive KV caches in VRAM for extended periods. This creates a true test of the system's memory management, prefix caching eviction policies, and overall concurrent capacity limits under severe, sustained memory pressure.
- Note on context management (the sliding window): To prevent exceeding the 128,000-token boundary during these long, 20+ turn sessions, the load generator employs a sliding window mechanism. The client permanently pins the system prompt and the massive initial code base payload (submitted in the first user turn) at the top of the context. It then uses a sliding window to retain only the 5 most recent conversation turns, dropping older messages from the middle. For example, by turn 7, the prompt sent to the engine consists of the permanently pinned code base payload, followed immediately by the recent conversational history (turns 3 through 7). The intermediate messages (turn 2) are silently evicted from the middle of the context. This perfectly mirrors how real-world AI IDEs (such as Cursor and GitHub Copilot) and advanced agentic frameworks manage context limits during extended debugging sessions, ensuring the engine is tested against realistic context-pruning behavior while never losing the foundational code base.

Crucially, the P2 contexts are not forced to a single, identical length. The dataset builder assigns each P2 session one of several distinct synthetic monorepos, and then pads the remainder of the prompt with variable-length unique context, simulating different local file modifications. This ensures the initial high-context payloads naturally vary in size across sessions. A degenerate distribution, where every P2 prompt is exactly the same massive length and shares the exact same prefix, makes prefix caching trivially effective and severely understates the real prefill pressure. By introducing this natural variance in context sizes and shared roots, the workload forces the engine to handle a realistic distribution of memory allocations and prefill bursts, accurately simulating the chaotic memory pressure of an enterprise environment.

This geometry ensures that the vLLM prefix cache is heavily utilized for the shared roots, while the unique local context forces the GPU to perform realistic prefill compute for each request. This stands in stark contrast to naive benchmarks that rely on the homogeneous prompt assumption, a scenario where testers send identical massive prompts repeatedly, resulting in an artificial 100% cache hit rate that bypasses prefill compute entirely and inflates performance metrics. By balancing shared infrastructure with isolated, unique feature development, our approach provides a highly realistic simulation of enterprise memory pressure.

Figure 1: Initial Turn Dataset Geometry

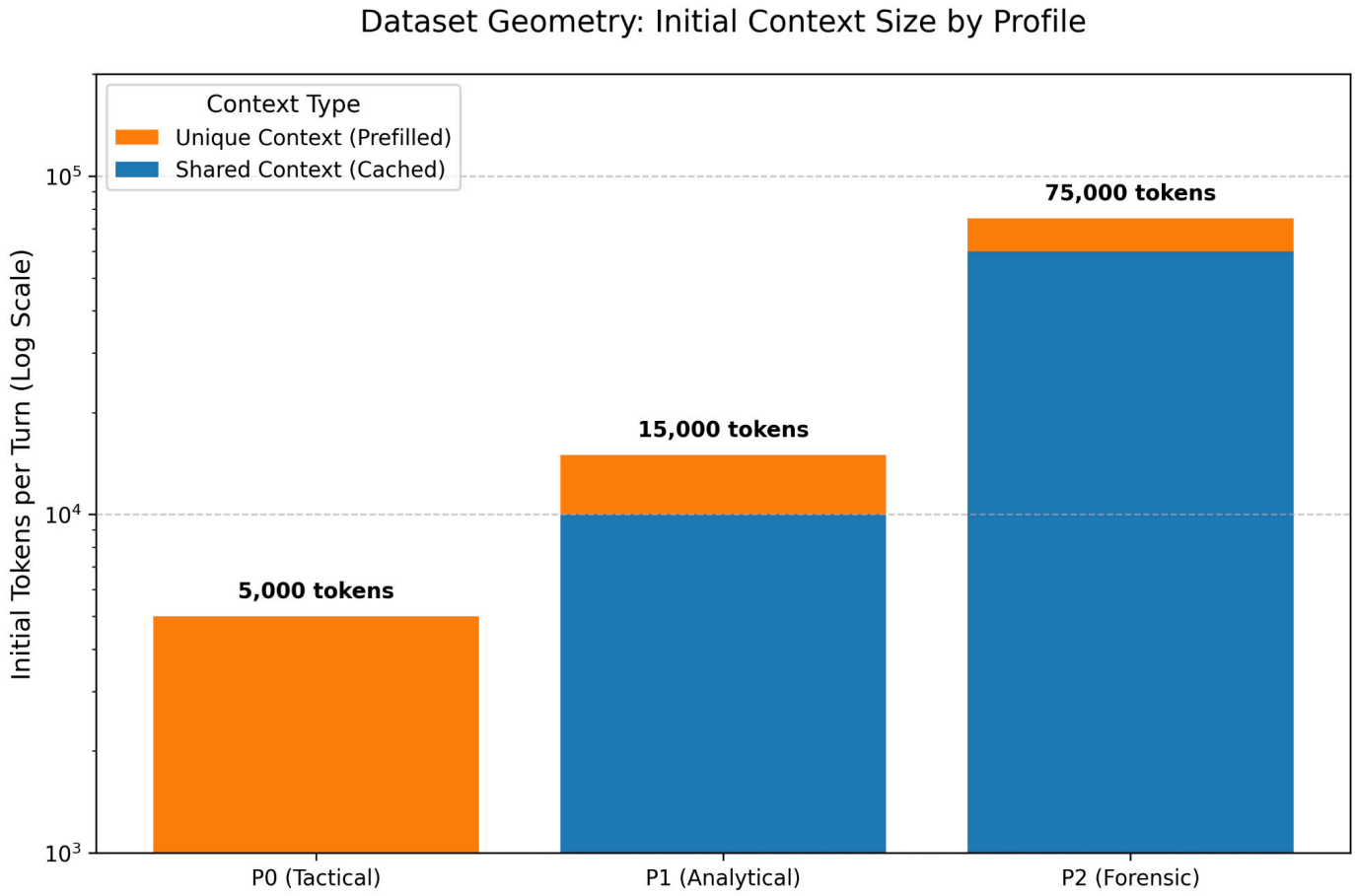


Figure 1 illustrates the token distribution at the start of a session (turn 1) for each profile using a logarithmic scale on the y-axis to clearly illustrate the massive disparity between P0/P1 and P2 payloads. As the simulated developers engage in multi-turn conversations, these initial baseline contexts grow organically as new user messages and model responses are appended to the history.

4.4 Preventing Prefix Cache Root Divergence

To ensure the vLLM radix tree can effectively cache the shared common ground, the order of elements within the prompt is critical. vLLM prefix caching operates by matching token sequences starting from the absolute beginning (token 0) of the prompt.

If a unique identifier, such as a session ID, timestamp, or user-specific metadata, is injected at the very beginning of the system prompt, it causes the radix tree to branch immediately. Consequently, even if an identical 80,000-token shared monorepo block follows, the inference engine cannot deduplicate it because the parent nodes in the cache tree are different. This phenomenon, known as root divergence, completely neutralizes the memory and compute benefits of prefix caching.

To prevent this, our workload enforces strict `PrefixCacheOrdering`. The massive shared context is always placed at the absolute beginning of the payload. Any unique session identifiers or cache-busting elements are appended to the end of the first user turn. This structural guarantee ensures that the massive prefixes remain byte-for-byte identical across different user sessions, allowing vLLM to achieve realistic cache hit rates while preserving the uniqueness of individual queries.

This design decision is not merely an artifact of the benchmark harness; it reflects the industry-standard best practice for interacting with modern LLM inference engines. Major API providers, including OpenAI and Anthropic, explicitly instruct developers to place static, shared context, such as large documents, core instructions, or tool definitions, at the absolute beginning of the prompt, and dynamic, user-specific content at the end. By adhering to this `PrefixCacheOrdering`, our workload accurately simulates how a well-architected enterprise application would format its requests to maximize cache hit rates and minimize latency.

4.5 The P2 Lazy-Generation Problem and Fix

During initial development, P2 requests consistently produced only 1 to 15 tokens before stopping. When the `gpt-oss-120b` model encounters a large context, it enters an analysis path and emits a special tool-call token: `<|call|>`. Because the benchmark harness is not an interactive agent, there is no tool executor listening for the call, and the response simply stops.

It is important to note that this lazy generation behavior is a specific quirk of the `gpt-oss-120b` model, likely stemming from its specific tool-use training and chat template. Other LLMs may not exhibit this issue when presented with massive contexts.

For `gpt-oss-120b`, the fix is to append an empty assistant message to the conversation and set `continue_final_message=True` in the API request body. This forces the model to bypass the tool-call path and output its analysis directly into the final output channel.

Chapter 5: Measurement Methodology and Stepped Optimization

5.1 Service-Level Agreement Definition

The SLA used throughout this work defines two independent thresholds:

- Time-to-first-token (TTFT): Below 10,000 ms (p95)
- Inter-token latency (ITL): Below 200 ms (p95)

These thresholds are grounded in human-computer interaction (HCI) research and industry best practices for streaming LLM applications:

- ITL and the HCI comfort band: The average adult reading speed is approximately 250 words per minute, which translates to roughly 5 to 7 LLM tokens per second (or 140 ms to 200 ms per token). While an ITL of 100 ms (10 tokens per second) is often cited as an ideal target, human readers remain comfortable as long as the text generation outpaces their reading speed. Therefore, we define an HCI comfort band with a ceiling of 200 ms per token. A p95 ITL of 200 ms guarantees a delivery rate of at least 5 tokens per second, ensuring that the text appears on the screen faster than the user can read it, providing a smooth, continuous stream without perceived stuttering or lag.
- TTFT: While simple chatbots often target sub-second TTFTs, complex AI coding assistants processing massive contexts (up to 128,000 tokens) operate under different user expectations. According to Nielsen's usability heuristics, response times between 1 second and 10 seconds are acceptable for keeping a user's attention focused on the dialogue, provided they understand the system is performing a heavy computational task. A strict 10,000-ms (10-second) p95 TTFT strikes a pragmatic balance: it is fast enough to maintain the developer's cognitive flow while allowing the inference engine sufficient time to process massive, multi-file repository contexts.

To ensure consistency and prevent configuration drift, these SLA thresholds are strictly enforced via environment variables (`SLA_MAX_TTFT_MS` and `SLA_MAX_ITL_MS`) acting as a single source of truth across all benchmarking and optimization scripts.

Mechanically, TTFT is dominated by prefill time and scheduler queueing, while ITL measures the wall-clock gap between consecutive tokens:

ITL ≈ t_step + t_sched

where:

t_step = GPU time for one decode forward pass

t_sched = CPU-side scheduling overhead

5.2 Capacity Discovery and Parameter Optimization

Instead of running wide, arbitrary capacity sweeps across a massive range of users, we employ a strictly phased approach governed by a monotonic failure rule: *Under a fixed workload mix and fixed vLLM configuration, if the SLA fails at concurrency N, we do not treat higher counts (N + Δ) as plausible operating points without changing configuration.*

In our protocol, offered load and scheduler pressure only rise with N, so continuing upward after a failure would mostly burn GPU time. This is the ordinary queueing intuition that steady-state latency does not improve when you add concurrent sessions while holding the engine setup constant. Readers who prefer a formal concurrency-scaling framing may relate the same idea to Gunther's Universal Scalability Law and related work; the sweep logic itself does not depend on that literature.

Because queuing theory dictates that latency strictly increases as the inference engine takes on more concurrent requests (due to contention and scheduling overhead), there is no performance recovery at higher loads. Therefore, our capacity planning methodology follows a strict progression:

1. Capacity discovery and headroom allocation: We test increasing concurrency levels sequentially (150, 175, 200 users). At each step, we run a short Optuna study (introduced in Chapter 6) with an early-exit condition. If any trial passes the SLA, the sweep immediately halts the study, records a PASS, and steps up to the next concurrency level. Crucially, the vLLM instance is kept running between these concurrency steps. Restarting the engine would invalidate the test by clearing the prefix cache; preserving the engine state ensures the test accurately reflects the cache warmth and memory fragmentation of a continuously running production server. If all trials fail, it records a FAIL and halts the entire sweep. This efficiently pinpoints the absolute capacity ceiling. From this ceiling, we select a target operational concurrency that leaves a safe margin (headroom) to absorb unpredictable traffic spikes.
2. Parameter optimization (the exhaustive phase): Once the target operational concurrency is selected, we lock the concurrency to that specific user count and run a full, exhaustive 50-trial Optuna NSGA-II optimization. This gives the evolutionary algorithm enough time to fully explore the Pareto front and find the absolute optimal configuration that better satisfies SLAs at that specific load.
3. The golden run and telemetry analysis: Optuna trials are inherently short (typically 15 to 20 minutes) to allow for many iterations. However, short tests can mask slow-building issues like memory fragmentation, thermal throttling, or subtle scheduler deadlocks. To validate the final capacity, we take the single best configuration from Step 2 (the exhaustive phase) and run a continuous 60-minute golden run soak test, capturing deep hardware telemetry to prove why the system remains stable. The capacity is only considered validated if the system maintains the SLA for the full hour.

Chapter 6: Parameter Optimization with NSGA-II

6.1 Problem Formulation

The vLLM scheduler exposes several parameters that jointly determine throughput and latency. No single parameter dominates all objectives simultaneously. This is a multi-objective optimization problem:

1. Maximize generation throughput (tokens per second: tok/s)
2. Minimize p95 TTFT (ms)
3. Minimize p95 ITL (ms)

Once the capacity discovery phase identifies the ceiling and we select our target operational concurrency, we conduct a targeted, exhaustive Optuna NSGA-II study only at that specific target concurrency.

6.2 NSGA-II and Pareto Optimality

What is a Pareto-optimal solution? A configuration is Pareto-optimal if there is no other configuration that is strictly better on at least one objective without being worse on any other. In other words, every Pareto-optimal configuration represents a genuine trade-off: to get more throughput, you must accept higher ITL; to get lower ITL, you must accept lower throughput. The set of all Pareto-optimal configurations forms the *Pareto front*: the efficient frontier of the search space. For an infrastructure engineer deciding how to configure the serving stack, the Pareto front is the only set of configurations worth considering: any configuration not on the front is dominated by at least one that is better in every relevant dimension.

Why not a grid search? With multiple parameters at several values each, a complete grid contains hundreds of combinations. At approximately 10 minutes per trial, a full grid search would require days of continuous GPU time. More importantly, a grid search evaluates all combinations uniformly, spending as much time on configurations that clearly violate the SLA as on configurations near the Pareto front.

NSGA-II is a population-based evolutionary algorithm specifically designed for multi-objective optimization. It works as follows:

1. Non-domination ranking: Configurations are evaluated and ranked. The core idea is best illustrated with a small example. Suppose three configurations (X, Y, Z) have been evaluated on two objectives (throughput and ITL):
   - X achieves 6000 tok/s and 30-ms ITL.
   - Y achieves 5500 tok/s and 35-ms ITL.
   - Z achieves 5800 tok/s and 25-ms ITL.

   Y is *dominated* by X because X is better on both objectives: higher throughput and lower ITL. However, X and Z do not dominate each other: X has higher throughput, but Z has lower ITL. Neither is strictly better on *all* objectives, so both are non-dominated. NSGA-II assigns an integer *rank* to each configuration based on this logic. All non-dominated configurations receive rank 1 (the Pareto front). After removing rank 1, the algorithm repeats the process on the remaining configurations: the next set of non-dominated configurations receives rank 2, and so on.
2. Crowding distance sorting: Within each rank, configurations are sorted by *crowding distance*. For each configuration, the algorithm looks at its two nearest neighbors along each objective axis and sums the gaps. A configuration that sits in a sparse region of the objective space (far from its neighbors) gets a high crowding distance; one that is tightly clustered gets a low crowding distance. The algorithm favors high crowding distance to preserve diversity across the Pareto front and prevent convergence to a single cluster.

3. Evolutionary breeding: The next generation of configurations to evaluate is produced in three steps:
   a. Tournament selection: Pairs of configurations are compared, and the one with the better rank (or, if tied, the higher crowding distance) is chosen as a parent.
   b. Simulated binary crossover: Two parent configurations are blended to produce a child configuration whose parameter values sit somewhere between the parents.
   c. Polynomial mutation: Small random perturbations are applied to the child's parameter values to maintain diversity and prevent the search from getting stuck in a local optimum.

This evolutionary process allows the optimizer to find the absolute optimal configuration (the miracle setup of `MAX_NUM_BATCHED_TOKENS`, `MAX_NUM_SEQS`, and `GPU_MEMORY_UTILIZATION`) that maximizes throughput while strictly adhering to the SLAs, without wasting time on mathematically inferior setups.

To understand the impact of this optimization, it is crucial to examine how each of these three core vLLM parameters governs the engine's performance:

- `MAX_NUM_BATCHED_TOKENS` (default:8192): This parameter determines the absolute ceiling on the number of tokens (both prompt and generation) processed in a single forward pass of the model. In a chunked prefill architecture, this parameter is critical for balancing throughput and responsiveness. A higher limit allows the engine to ingest larger chunks of prompts simultaneously, maximizing GPU compute utilization (throughput). However, if set too high, massive prefill operations can monopolize the GPU, starving ongoing generations and causing severe spikes in TTFT and ITL. Furthermore, excessively large values (such as 8192) can cause catastrophic CUDA graph compilation timeouts during engine initialization for massive MoE models. Our Optuna sweep identified `batched_tokens: 2048` as the optimal setting to prevent these compilation timeouts while maintaining strict SLAs.
- `MAX_NUM_SEQS` (default: 1024) This defines the maximum number of concurrent requests the vLLM scheduler is allowed to process in a single batch. This parameter directly governs the concurrency ceiling of the inference engine. Increasing this value allows the system to multiplex more users simultaneously, improving overall batch efficiency and throughput. Conversely, pushing it too high spreads the GPU's attention bandwidth too thin, inevitably degrading ITL as the engine struggles to generate the next token for hundreds of sequences at once.
- `GPU_MEMORY_UTILIZATION` (default: 0.90): This dictates the fraction of total physical VRAM pre-allocated by vLLM upon startup for model weights and the KV cache. Because the KV cache stores the state of all active conversations, maximizing this parameter, at 0.90 or 0.95, provides the largest possible memory pool for concurrent users and long context windows. However, setting it too close to 1.0 leaves insufficient headroom for PyTorch's dynamic memory allocations during forward passes, risking catastrophic out-of-memory (OOM) crashes under peak load.

Chapter 7: Results: 4x NVIDIA H100 (80GB)

7.1 Stepped Capacity Search Progression

The discovery phase on the 4x H100 VM with the default vLLM launch parameters revealed a highly efficient system capable of scaling well beyond initial expectations. The system successfully passed successive 15-minute probes at increasing concurrency levels, pushing the absolute ceiling higher as detailed in Table 1.

Table 1: Stepped Capacity Discovery (4x H100)

Concurrency	p95 TTFT (ms)	p95 ITL (ms)	SLA Status
150	1933.44	122.15	PASS
175	325.51	103.22	PASS
200	319.10	104.06	PASS
225	324.63	115.16	PASS
250	288.76	115.51	PASS
275	425.56	114.89	PASS
300	1525.54	143.22	PASS

At 300 users, p95 TTFT is higher than at the intermediate steps (queueing and prefill competition), while p95 ITL remains comfortably inside the SLA. We still record a PASS because both metrics stay under their respective ceilings; the golden run in Section 7.2, The 60-Minute Golden Run (300 Users) confirms latency under the tuned configuration.

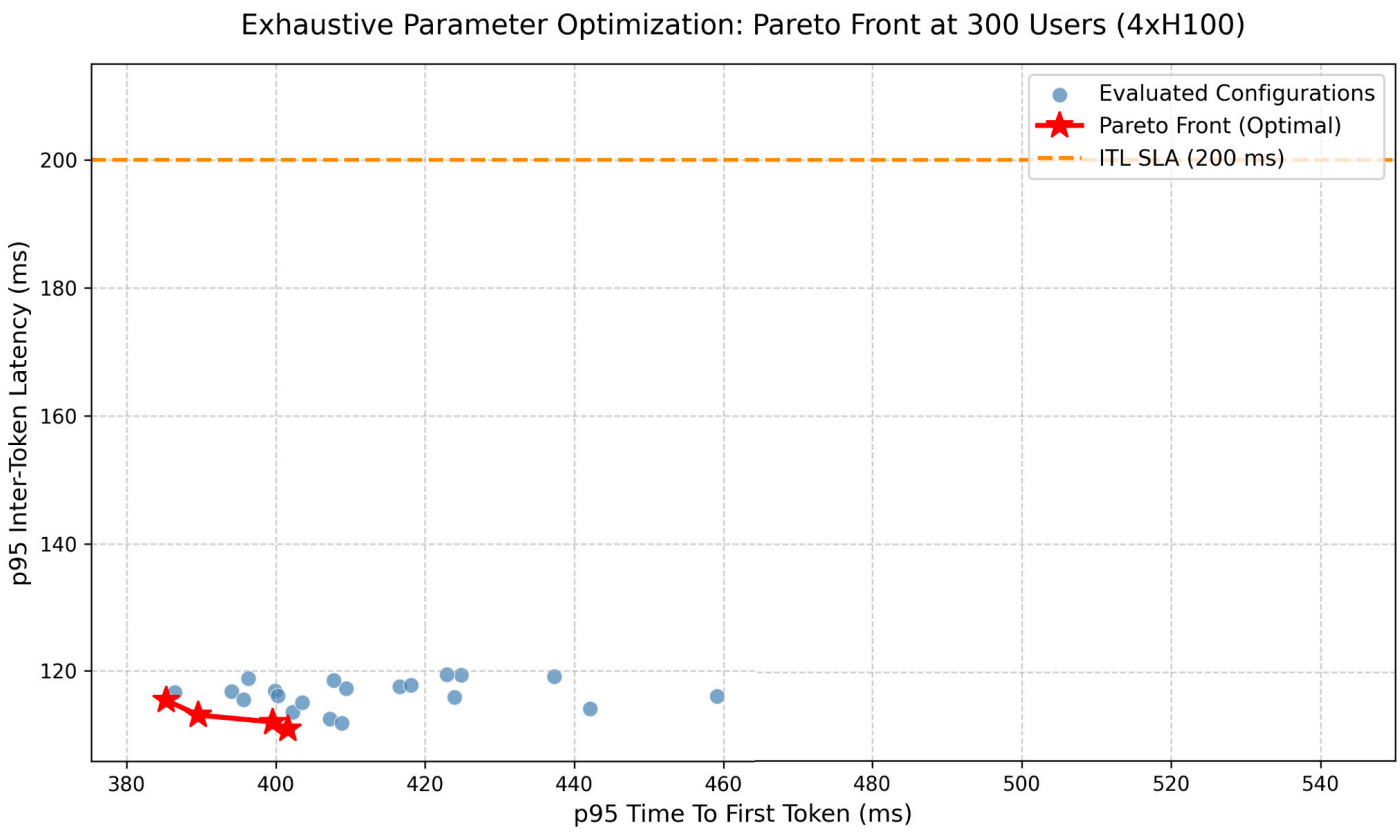
From this discovery phase, we selected 300 users as our target operational concurrency. Even though the system successfully passed the SLA at 300 users (with an unoptimized baseline p95 ITL of 143.22 ms) and could likely scale to higher concurrencies, 300 concurrent users represents the maximum expected peak load in our real-world enterprise environment. We deliberately chose not to push the system higher without a real need, establishing 300 users as our operational target to ensure we retain a safe margin of headroom to absorb unpredictable traffic spikes that deviate from the standard P0/P1/P2 distribution.

Once this target was locked, an exhaustive Optuna NSGA-II sweep successfully mapped the Pareto front at exactly 300 users. It identified `batched_tokens: 2048` as the optimal setting to prevent CUDA graph compilation timeouts while maintaining strict SLAs and protecting the decode phase from massive prefill starvation.

Table 2: Exhaustive Pareto Front at Target Concurrency (300 Users)

Trial	p95 TTFT (ms)	p95 ITL (ms)	MAX_NUM_BATCHED_TOKENS	MAX_NUM_SEQS	GPU_MEMORY_UTILIZATION
53	385.28	115.40	2048	384	0.94
52	389.61	113.10	2048	512	0.90
41	399.58	112.03	2048	384	0.93
57	401.60	110.93	2048	256	0.90

Figure 2: The Pareto Front Discovered by the NSGA-II Algorithm during the Exhaustive Phase at 300 Concurrent Users



As shown in Figure 2, the red stars represent the non-dominated optimal configurations, while the blue dots represent the broader search space explored by the algorithm. All optimal configurations sit safely below the 200 ms ITL SLA.

7.2 The 60-Minute Golden Run (300 Users)

To definitively prove stability at our chosen target concurrency, we locked the system to 300 users and executed a continuous 60-minute golden run soak test. The results were definitive:

- Concurrency: 300 users
- Completion Rate: 100.00%
- p95 TTFT (All): 301.10 ms (SLA: below 10,000 ms)
- p95 ITL: 58.18 ms (SLA: below 200 ms)
- OVERALL SLA: PASS

The system comfortably sustained 300 concurrent simulated developers. The p95 ITL of ~58 ms translates to a delivery rate of over 17 tokens per second (roughly 765 words per minute), providing an exceptional, stutter-free reading experience well within the HCI comfort band.

7.3 Hardware and Software Profiling at Target Concurrency

Profiling at the 300-user target revealed the following bottlenecks and utilization metrics. A clean NSYS hardware trace was captured during active load, confirming the kernel distribution:

Figure 3: CUDA Kernel Distribution

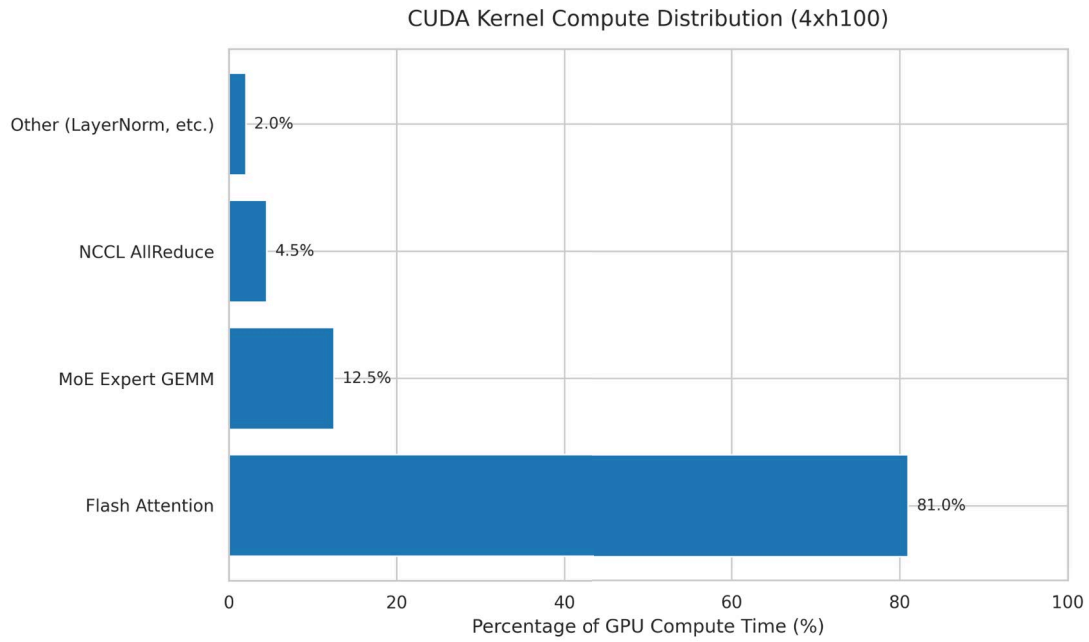


Table 3: Hardware and Software Profiling at Target Concurrency (4x H100)

Metric Category	Metric Name	Value
Software (vLLM)	Prefix Cache Hit Rate	86.88%
	Peak Requests Waiting	60
	Active KV Cache Usage	10.5%
Hardware (DCGM)	SM Active	36.9%
	DRAM Active	15.1%
	NVLink Bandwidth	0.51 GB/s
Compute (NSYS)	Top CUDA Kernel Category	Attention (81.0%)

NOTE: Regarding telemetry: Exporter NVLink metrics (including `DCGM_FI_DEV_NVLINK_BANDWIDTH_TOTAL`) can read as zero in some deployments (see NVIDIA/dcgm-exporter#316), so our monitoring stack explicitly aggregates the raw `DCGM_FI_PROF_NVLINK_TX_BYTES` and `DCGM_FI_PROF_NVLINK_RX_BYTES` counters to measure interconnect saturation. This ensures we capture the real NVLink data (0.51 GB/s) rather than a false zero.

Analysis: The system is heavily compute-bound by attention kernels (81.0%), which is expected under high prefix cache hit rates (86.88%). The high cache hit rate indicates excellent memory efficiency: the active KV cache usage remains incredibly low because the massive shared monorepos are successfully deduplicated in VRAM, and the client-side sliding window prevents unbounded context growth. The low SM active (36.9%) and DRAM active (15.1%) figures suggest the bottleneck is per-step duration (ITL) rather than aggregate GPU saturation. The system is pacing itself sufficiently to maintain the strict ITL SLA.

7.4 The Value of Parameter Optimization

To illustrate the critical importance of the NSGA-II optimization phase, it is instructive to compare the vLLM default parameters against the optimal configuration discovered for the 4x H100 system at 300 users:

- `MAX_NUM_BATCHED_TOKENS`: Reduced from the default 8192 down to 2048. This was the most consequential change. While 8192 maximizes theoretical throughput, it allowed massive P2 prefill chunks to monopolize the GPU, driving the p95 ITL well past the 200-ms SLA and causing CUDA graph compilation timeouts. Capping it at 2048 forced the engine to interleave smaller prefill chunks with decode steps, protecting the streaming experience.
- `MAX_NUM_SEQS`: Reduced from the default 1024 down to 384. The default allows the scheduler to multiplex too many concurrent requests, spreading the GPU's attention bandwidth too thin and degrading ITL. Capping it at 384 aligned the scheduler's ambition with the hardware's actual compute limits.
- `GPU_MEMORY_UTILIZATION`: Increased slightly from the default 0.90 to 0.94. Because the system was compute-bound rather than memory-bound, allocating more VRAM to the KV cache provided a small buffer for peak concurrency spikes without risking OOM errors.

Without this optimization phase, the system would have failed the ITL SLA at 300 users under the default configuration. The Optuna sweep mathematically proved that sacrificing some theoretical maximum throughput by lowering batch sizes)was mandatory to satisfy the strict tail-latency requirements of a stateful, massive-context workload.

Below is the exact vLLM launch command used for the 4x H100 golden run, incorporating the optimal parameters discovered by the NSGA-II sweep, along with the baseline configuration flags required for this specific MoE model:

Shell

```
VLLM_FLASHINFER_ALLREDUCE_FUSION_THRESHOLDS_MB='{"4":32}' \
vllm serve /model/gpt-oss-120b \
  --tensor-parallel-size 4 \
  --max-num-batched-tokens 2048 \
  --max-num-seqs 384 \
  --gpu-memory-utilization 0.94 \
  --enable-prefix-caching \
  --enable-chunked-prefill \
  --async-scheduling \
  --kv-cache-dtype fp8 \
  --api-server-count 4 \
  --compilation-config '{"cudagraph_mode":"PIECEWISE","cudagraph_capture_sizes":[2048]}' \
  --disable-log-requests \
  --port 8000
```

Chapter 8: Results: 2x NVIDIA H200 (141GB)

8.1 The 60-Minute Golden Run (85 Users)

The 2x H200 system underwent a similar rigorous testing protocol. After initial capacity discovery probes, we selected 85 concurrent users as our target operational concurrency and executed a 60-minute golden Rrn soak test.

- Concurrency: 85 users
- Completion Rate: 100.00%
- p95 TTFT (All): 1063.29 ms (SLA: below 10,000 ms)
- p95 ITL: 152.09 ms (SLA: below 200 ms)
- Active KV Cache Usage: 3.9%
- NVLink Bandwidth: 7.87 GB/s
- OVERALL SLA: PASS

While the system successfully passed at 85 users, it failed to scale to the 100+ user mark during longer soak tests due to ITL degradation.

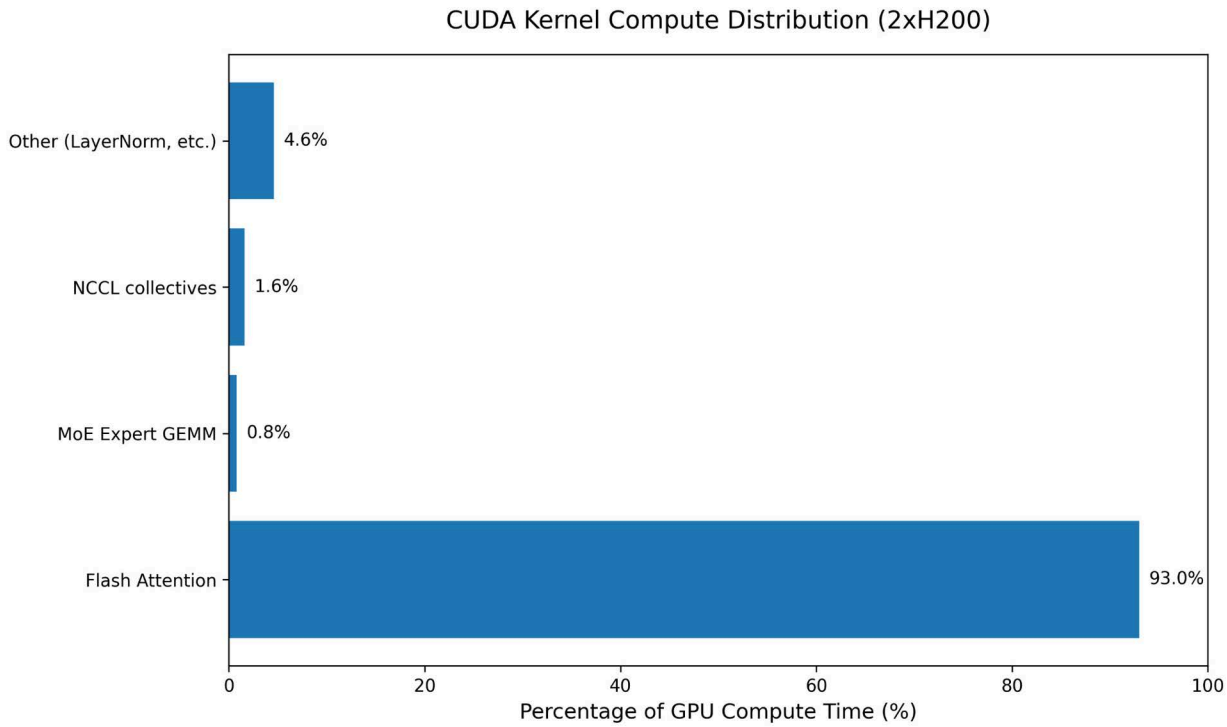
8.2 Kernel-Level Compute Distribution (NVIDIA Nsight Systems)

At the same 85-user target concurrency used for the golden run, we captured a short NVIDIA Nsight Systems window on the live `vllm serve` process (direct `nsys profile` wrapper, not sidecar attach) once the CUDA graphs and load had stabilized. Kernel time was rolled up from `nsys stats` using the `cuda_gpu_kern_sum` summary into four buckets: all `FlashAttn*` variants as Flash Attention, `nvjet_*` expert GEMM templates as MoE Expert GEMM, all `nccl*` kernels as NCCL collectives, and everything else as Other.

Table 4: NSYS Kernel Time Share (2x H200, 85 Users)

Category	Percent of GPU Time
Flash Attention	93.0%
MoE Expert GEMM	0.8%
NCCL collectives	1.6%
Other (LayerNorm, KV cache helpers, etc.)	4.6%

Figure 4: CUDA Kernel Distribution, Table 4 Data



Compare Figure 4 to Figure 3 (4x H100 at 300 users): Flash Attention rises from 81.0% to 93.0%, while MoE Expert GEMM falls from 12.5% to 0.8% and NCCL from 4.5% to 1.6% (the H100 percentages use the same four-way `nsys stats` kernel bucketing described for H200 in Section 8.2, Kernel-Level Compute Distribution (NVIDIA Nsight Systems)). The platforms are not direct comparisons on concurrency (300 users versus 85 users) or tensor-parallel width (TP = 4 versus TP = 2), but the contrast is still instructive. On 2x H200 at this operating point, a larger fraction of wall-clock time is spent inside attention kernels relative to expert matmuls; with only two GPUs, tensor-parallel collectives are a smaller share of the trace than on four-way H100, while the higher per-GPU HBM bandwidth of H200 still leaves attention as the dominant cost center. Together with the lower `max_num_batched_tokens` (1024 vs 2048) and stronger prefill/decode interleaving effects discussed below, this profile explains why tail ITL, not average GPU occupancy, is the limiting factor when approaching the concurrency wall.

8.3 Hardware Scaling Analysis: Why the Nonlinear Drop?

The scaling from a 2x H200 system (85 users) to a 4x H100 system (300 users) is highly nonlinear (~3.5x concurrency rather than the expected 2x) when bound by strict tail-latency SLAs. This is driven by three primary architectural factors:

1. Aggregate memory bandwidth (the decode bottleneck): LLM decode generation is heavily memory-bandwidth bound. The 4x H100 (SXM) provides ~13.4 TB/s aggregate bandwidth, while the 2x H200 provides ~9.6 TB/s. The 4x H100 system does not just have 2x the compute; it has ~40% more aggregate memory bandwidth. This allows it to read weights and KV cache significantly faster per forward pass, keeping the baseline ITL lower and leaving more headroom to absorb larger batch sizes before hitting the 200-ms ceiling.
2. Tensor parallelism (TP = 4 versus TP = 2) math division: In a YTP setup, matrix multiplications are divided across GPUs. At TP = 4, each GPU computes one quarter of the math per layer. At TP = 2, each GPU computes one half of the math. Because the 2x H200 system does twice the math per GPU per token, its baseline compute time per forward pass is longer. To stay under the strict 200-ms ITL SLA, the vLLM scheduler is forced to cap the batch size much earlier on the TP = 2 system.
3. The chunked prefill penalty (software compounding): Because TP = 2 does twice the math per GPU, `max_num_batched_tokens` had to be reduced from 2048 to 1024 on the 2x H200 to prevent SM lockup during heavy prefill phases. On the 4x H100, a 4000-token prompt is processed in two chunks. On the 2x H200, that same prompt requires four chunks. Since vLLM interleaves prefill chunks with decode steps, the 2x H200 system requires twice as many scheduling interruptions to process the same incoming prompt. These frequent interruptions cause severe jitter in the decode phase, driving up the p95 ITL (tail latency). The 2x H200 system fails the SLA at higher concurrencies not because its average ITL is too high, but because the prefill jitter pushes the 95th percentile over the 200-ms limit.

Chapter 9: Thermal Stability Under High Load

9.1 The Relevance of Thermal Monitoring in Tensor Parallelism

When utilizing tensor parallelism (TP), the model's weights and matrix multiplications are sharded across multiple GPUs. During each forward pass, these GPUs must synchronize their intermediate results using AllReduce operations over NVLink.

Because of this strict synchronous communication barrier, the entire inference step can only proceed as fast as the slowest GPU in the group. If a single GPU experiences thermal throttling, its streaming multiprocessor (SM) clock speed will dynamically drop to shed heat. This underperforming GPU immediately becomes a straggler, forcing the other healthy GPUs to wait idly at the NVLink synchronization barriers. Consequently, a thermal issue on just one GPU degrades the throughput and ITL of the entire multi-GPU system.

9.2 Case Study: The Impact of a Single Thermal Straggler

To illustrate this, we recovered thermal data from a previous test run on an 8×H100 physical host (a different machine from the 4×H100 VM that is the primary system under test in Chapter 3 and Chapter 7), where the benchmark severely underperformed expectations. The data for the first four GPUs of that 8-GPU run clearly shows that a single faulty GPU is bottlenecking the entire group:

Table 5: Add Table Title Here

GPU	Avg. Temp. (°C)	Max. Temp. (°C)	Avg. SM Clock (MHz)	Thermal Throttling
GPU 0	~40.0	45.0	~1590 to 1980	No
GPU 1	~40.0	45.0	~1590 to 1980	No
GPU 2	~40.0	45.0	~1590 to 1980	No
GPU 3	~86.0	86.0	below 900 (throttled)	Yes (SW thermal slowdown)

The healthy GPUs maintained temperatures between 35°C and 45°C under load, operating within their expected base (~1590 MHz) and boost (~1980 MHz) clock ranges. Meanwhile, GPU 3 immediately hit the 86°C thermal limit and remained there, triggering the hardware thermal slowdown. Power draw data confirmed GPU 3 was choked at 238W (well below the 700W limit), while the healthy GPUs were pulling 300W to 350W. This forced a >50% drop in SM clock speed on the throttled GPU, causing severe latency spikes across the entire tensor-parallel group. This demonstrates why monitoring thermal stability is a mandatory prerequisite before concluding that a system has reached its absolute compute or memory ceiling.

Chapter 10: Discussion

10.1 Why GPU Utilization Is Not an SLA Metric

A natural reaction to the ~37% GPU SM active figure at the capacity ceiling is that nearly 60% of the GPU's compute capacity is being left on the table. However, GPU utilization and SLA compliance are not correlated in the direction that intuition suggests.

GPU utilization measures aggregate occupancy over a time window: the fraction of SMs that had at least one active warp. It does not measure, and is not proportional to, the time required to complete one forward pass. The SLA, however, is defined in terms of per-step latency: ITL is the wall-clock time between consecutive tokens, which equals the duration of one decode forward pass. That duration is determined by the batch size, the model architecture, and the kernel implementations, none of which change when the GPU is made busier by filling scheduling gaps.

In practice, adding more work between forward passes (by increasing concurrency or widening the prefill chunk) increases throughput and GPU utilization, but it does not speed up the decode step and may slow it down if the additional work competes for GPU resources within the same step. For infrastructure engineers, the practical implication is that GPU utilization should be monitored as a cost-efficiency metric, not as an SLA-compliance metric. A system running at 40% GPU utilization that meets the SLA is correctly configured. Tuning to reach 95% utilization without verifying that ITL and TTFT remain within bounds will degrade the user experience.

10.2 The P2 Profile as a System Stress Multiplier

The P2 profile (10% of users) carries the heterogeneous high-context workloads defined in Section 4.3, The Power-Law of Code Access and User Profiles: each session draws one of several 60,000-token shared monorepos plus variable-length unique local context on turn 1 (the ~75,000-token initial geometry in Figure 1, varying across sessions rather than a single fixed ceiling), with sliding-window pruning and long-session growth toward the 128,000 limit. That footprint has an outsized effect on system behavior relative to population share. A single P2 request occupies tens of thousands of KV cache blocks for the duration of its generation.

Despite this, KV cache utilization remained highly efficient. The reason is two-fold: the partial common ground dataset allows massive deduplication of the shared monorepos, and human read-times space out the requests. This finding has a practical implication: for workloads with a similar P2 fraction and high prefix cache hit rates, KV cache size is not the binding constraint. Infrastructure engineers can allocate less VRAM to the KV cache (lower `GPU_MEMORY_UTILIZATION`) without impacting capacity, freeing headroom for other uses.

The P2 profile does, however, affect TTFT for P0 and P1 users. When a P2 prefill is in progress, the chunked prefill scheduler must interleave P2 prefill chunks with P0/P1 decode steps. Without chunked prefill, a single P2 request would monopolize the GPU for the entire duration of its prefill, stalling all concurrent decode operations. Chunked prefill bounds the per-step latency impact on other users.

Chapter 11: Conclusions

This paper presents a stateful, profile-based benchmarking framework for LLM inference capacity planning, validated on 4x H100 and 2x H200 servers running `gpt-oss-120b` under vLLM v0.16.0.

The central contribution is a methodology that treats capacity planning as a multi-objective optimization problem: finding the configuration that maximizes throughput while strictly satisfying latency SLAs (TTFT and ITL). By using heterogeneous user profiles, stateful conversational accumulation, 60-minute soak testing, and dual telemetry, this approach reveals system behaviors that single-prompt, stateless benchmarks obscure.

Key findings include the following:

1. The HCI comfort band is the true constraint: A p95 ITL well within the 200-ms ceiling (as demonstrated in the 4×H100 golden run at ~58 ms, roughly 765 WPM in Section 7.2, The 60-Minute Golden Run (300 Users)) provides an exceptional user experience. Pushing for arbitrarily lower ITLs, for example below 50 ms, artificially restricts capacity without providing a proportional benefit to human readers.
2. Chunked prefill is a vital trade-off: To protect the ITL of ongoing generations from massive P2 prefill spikes, `MAX_NUM_BATCHED_TOKENS` must be carefully tuned (2048 for 4x H100, 1024 for 2x H200). This sacrifices some TTFT speed to maintain a smooth streaming experience.
3. Hardware scaling is nonlinear under tail-latency constraints: The 4x H100 system achieved ~3.5x the capacity of the 2x H200 system (300 users versus 85 users). This is due to the compounding effects of aggregate memory bandwidth, tensor parallelism math division, and the chunked prefill penalty on smaller GPU clusters.
4. VRAM is not always the bottleneck: With effective prefix caching of shared code bases and a client-side sliding window, active KV cache usage remained remarkably low (~10.5%). The system was entirely bound by attention compute (TFLOPS) and memory bandwidth, not VRAM capacity. This highlights a critical dependency: when the client-side agent framework smartly manages context, such as via strict prompt ordering and sliding windows, it maximizes the inference engine's prefix cache efficiency, unlocking significantly more capacity. Infrastructure teams should evaluate whether their agent frameworks are architected to exploit these engine-side memory features.

Infrastructure engineers will note several key takeaways:

1. Statefulness and dataset geometry matter: Testing with flat, stateless prompts relies on the homogeneous prompt assumption. Real enterprise workloads often follow a power-law distribution of shared vs. unique code. A partial common ground dataset geometry accurately simulates this, and enforcing strict `PrefixCacheOrdering` is mandatory to prevent root divergence and achieve realistic cache hit rates.
2. MoE arithmetic intensity defies simple rooflines: For sparse MoE models like `gpt-oss-120b`, the mismatch between dense memory loading (fetching all expert shards) and sparse compute breaks traditional arithmetic intensity models at large batch sizes. Empirical, kernel-level profiling is required to truly understand the bottleneck.
3. Phased optimization is essential: Arbitrary, wide capacity sweeps waste expensive GPU compute on mathematically impossible configurations. A capacity discovery phase quickly finds the ceiling to establish a safe target concurrency, while an exhaustive parameter optimization phase (using NSGA-II) at that specific target ensures the system can strictly satisfy SLAs under sustained load.
4. GPU utilization is not an SLA metric: High GPU utilization indicates cost efficiency, not user responsiveness. A system can run at 40% SM Active and meet the SLA, or run at 95% and fail catastrophically. ITL and TTFT are the only metrics that define the user experience.

5. Memory is not always the bottleneck: Even with 10% of users submitting massive 80,000 to 128,000 token contexts, KV cache utilization remained highly efficient due to prefix caching and human think-times. The system was fundamentally compute-bound by attention kernels, meaning `GPU_MEMORY_UTILIZATION` could be lowered to free VRAM without impacting capacity.
6. Thermal vulnerabilities in tensor parallelism: Under TP >1, the entire inference step proceeds only as fast as the slowest GPU. A single GPU experiencing thermal throttling (dropping its SM clock speed) will force all healthy GPUs to wait at NVLink synchronization barriers, causing severe, system-wide latency spikes. Holistic hardware monitoring is a mandatory prerequisite for capacity planning.

BROADCOM®