CounselGPT: A Study on Efficient Inference serving using llama.cpp on various Quantized LoRA Legal fine-tuned LLMs on Cloud

Vasavi Doddamani, Mathesh Thirumalai
Department of Computer Science and Engineering
University of California, Santa Cruz
vdoddama, mthiruma @ucsc.edu

***Abstract*—Large Language Models (LLMs) show strong potential for specialized tasks such as legal question answering, statute interpretation, and contract analysis. However, achieving efficient inference for domain-adapted models requires far more than fine-tuning, it demands optimized model formats, intelligent caching, and a scalable cloud-native serving stack.**

This study presents *CounselGPT*, an end-to-end legal-domain LLM inference system designed to evaluate efficient serving of quantized LoRA-adapted models using llama.cpp across heterogeneous GPU environments (L40, L4, V100). The system integrates a microservice-based architecture consisting of a FastAPI inference backend, semantic embedding service, Redis-based caching layer, frontend interface, and unified monitoring with Prometheus/Grafana. Deployments were carried out on both Google Cloud (GKE) and the Nautilus research cluster. CounselGPT is a research prototype and is not intended to provide legal advice or replace consultation with a qualified attorney.

We analyze model behavior under various quantization levels, caching strategies, and workload patterns (load, stress, spike, and soak tests). The project was conducted to maximize performance with the resources provided. The results highlight the trade-offs between cost, latency, throughput, GPU utilization, and operational complexity. This work provides practical guidance for building efficient legal-domain LLM or any fine-tuned LLM services in real-world cloud environments.

***Index Terms*—Large Language Models, LoRA, Kubernetes, Cloud Computing, llama.cpp, Semantic Caching, Inference, KV-Caching.**

I. Introduction

Our goal in this project is to build a practical, cloud-native legal assistant that can be inferred with constrained resources while preserving reasonable quality and transparency. We focus on three complementary aspects:

- **Efficient domain adaptation:** Fine-tuning open-source base models for legal question answering using parameter-efficient techniques.
- **Cost and resource-aware model serving:** A serving stack that can run end-to-end on GPU-only clusters, but opportunistically uses CPUs when available to improve latency and throughput.
- **Cloud-native deployment and observability:** A microservices architecture containerized with Docker and orchestrated on Kubernetes, with monitoring and benchmarking support suitable for both GKE and Nautilus.

A. Contributions

The main contributions of this work are:

- A LoRA-based fine-tuning setup for legal-domain LLMs, including dataset preprocessing, training, and export of quantized GGUF artifacts.
- A microservice-based LLM serving architecture with separate model-serving, embeddings, router/gateway, cache, and frontend components.
- Semantic caching to identify similar prompts and reduce load on our systems.
- Production-style Kubernetes deployments on both GKE (with CPU/GPU backends, HPA, and GCS-backed model storage) and Nautilus (with node affinity, CephFS, and DCGM GPU telemetry).
- An evaluation and observability setup with Prometheus metrics, Grafana dashboards, and benchmark scripts for latency, throughput, and resource usage.

II. Background and Motivation

A. Parameter-Efficient Fine-Tuning with LoRA

Full fine-tuning of modern LLMs is often prohibitively expensive: all model weights must be updated and stored separately for each downstream task. Low-Rank Adaptation (LoRA) addresses this by freezing the base model and injecting trainable low-rank matrices into selected weight matrices, reducing the number of trainable parameters by several orders of magnitude while maintaining quality [1]. This makes it feasible to specialize LLMs to legal tasks using limited GPU resources and to share small adapter checkpoints independently of the base model.

B. Efficient Inference with `llama.cpp`

To make local deployment and edge/server inference practical, `llama.cpp` provides a highly optimized C/C++ inference engine for transformer models, with support for CPU and GPU backends and quantization schemes such as Q8 [2]. Quantizing models to GGUF formats and running them through `llama.cpp` can yield substantial memory savings and improved latency on commodity hardware.

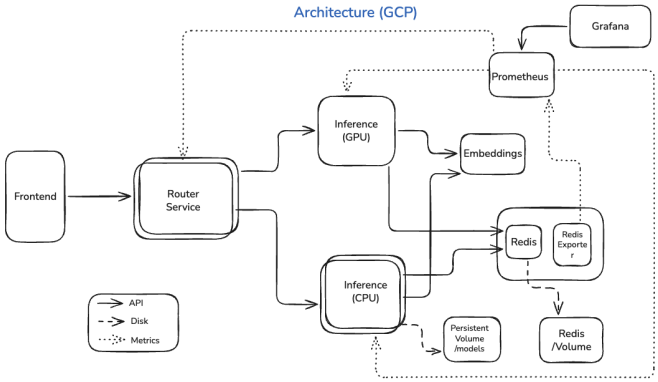

Fig. 1. Overall CounselGPT architecture on GKE, showing the router, CPU and GPU backends, Redis cache, monitoring stack, and external client access.

C. Cloud-Native Microservices and Observability

Best practices for cloud-native systems emphasize microservices, containerization, orchestration, autoscaling, and observability. Kubernetes, combined with tools such as Prometheus and Grafana, has become a standard platform for building scalable services with fine-grained monitoring of CPU, memory, and GPU utilization, as well as application-level metrics (latency, throughput, error rates). Our design follows these patterns to expose the behavior of the LLM stack under load and across different deployment environments.

III. Approach and System Design

A. Legal LoRA Training

For this project we adapt the open-source Qwen2.5-7B model to legal question answering using a parameter-efficient LoRA setup. We start from the public USLawQA dataset[1], which contains legal text entries extracted and organized from the United States Civil Code, and convert the examples into instruction–answer pairs where the answer includes an embedded chain-of-thought style rationale followed by the final conclusion, in the spirit of chain-of-thought prompting [4]. We then fine-tune only a small number of low-rank adapter weights on top of Qwen2.5-7B while keeping the original model parameters frozen, and finally convert the adapted model to GGUF format using `llama.cpp` tooling, including a Q8-quantized variant to reduce memory footprint and improve latency on commodity CPUs. The resulting model artifacts are stored in shared storage and loaded directly by the serving stack described in the next sections.

B. Initial Architecture and Constraints

The system initially used a straightforward architecture: the frontend sent HTTP requests directly to a single model-serving process running on a GPU node. This process loaded a quantized GGUF model via `llama.cpp`, ran inference synchronously, and returned the generated response. While this was sufficient for early experiments, two practical constraints emerged as we moved towards a more realistic deployment:

[1]https://huggingface.co/datasets/ArchitRastogi/USLawQA

- Due to quota limitations on GCP, only a single GPU node was available. This meant that GPU capacity was a scarce resource and had to be used carefully; any overload or downtime on that node would immediately affect all requests.
- We observed that many incoming requests were semantically similar or identical (e.g., repeated legal questions during testing and demonstrations), leading to repeated and unnecessary GPU work. Without caching, each such request would trigger a fresh forward pass, wasting GPU time that could have been avoided with a semantic cache.

These constraints motivated a redesign in which (i) CPU and GPU backends are treated as separate, first-class deployment targets, and (ii) a router and cache layer sit in front of the model-serving API to direct traffic and reuse prior results whenever possible.

1. Model-Serving Backend (API)

The core of the inference stack is a FastAPI application running on Uvicorn workers. Each backend instance (CPU or GPU) is responsible for serving two quantized GGUF models: a Qwen2.5-7B (8-bit) variant with LoRA adapters, and a Llama-2-7B (4-bit) variant used as an additional baseline. The Qwen2.5-7B model is eagerly loaded at startup so that the primary legal assistant is immediately available, while the Llama-2-7B model is loaded lazily on first use during inference calls to avoid unnecessary memory pressure when it is not needed.

Both models are configured via environment variables that specify model paths, context length, number of threads, and decoding parameters, and are exposed through a consistent HTTP interface with endpoints such as `/infer` that invoke the underlying `llama.cpp`-based runtime to generate tokens for each user prompt.

CPU and GPU backends share the same API shape so they can be swapped transparently by the router. The main difference lies in deployment configuration: GPU-backed instances request GPU resources from Kubernetes and run inside a CUDA-optimized container, while CPU-backed instances run entirely on commodity CPU nodes with more conservative concurrency settings.

2. Router and CPU/GPU Scheduling

To mediate access to the scarce GPU and to avoid overloading a single device, we introduce a lightweight Python router that acts as the single entry point for all client traffic. The router is responsible for CPU/GPU scheduling, request queuing, and graceful degradation when the GPU path is unavailable.

For each incoming request, the router first inspects any explicit flags (for example, a field indicating that the user prefers or requires CPU-only execution). If no such override is present, the router tries to send the request to the GPU backend, but only if several conditions are met:

- A circuit breaker for the GPU path is closed, indicating that recent failures have not exceeded a threshold.

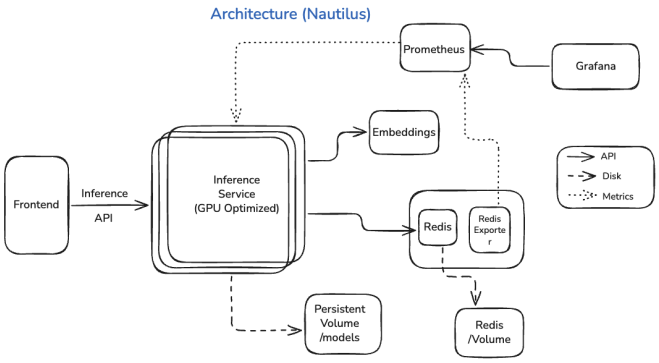

Fig. 2. CounselGPT deployment on the Nautilus cluster, highlighting GPU-only backends, scheduling preferences across GPU types, and shared monitoring and storage.

- The most recent health check for the GPU backend is positive.
- The internal GPU queue has remaining capacity.

If any of these checks fail, the router immediately routes the request to the CPU backend instead of overloading the single GPU node. This ensures that a temporary spike or failure on the GPU does not block the entire system; instead, traffic degrades gracefully onto CPU nodes with higher latency but continued availability.

To avoid unbounded concurrency on the GPU backend, the router maintains an explicit queue of GPU-bound requests. In simplified form, the routing logic can be expressed as:

```
if request.use_cpu:
    route_to_cpu()
elif circuit_allows_gpu() and
     gpu_healthy() and
     gpu_queue.has_capacity():
    enqueue_to_gpu(request)
else:
    route_to_cpu()
```

Dedicated worker tasks pull from the GPU queue and forward requests to the GPU backend API. If the queue is full or the circuit breaker is open, new requests are routed to CPU immediately. This explicit queuing provides backpressure and prevents the single GPU from becoming a bottleneck that causes long tail latencies for all users.

3. Semantic Caching

To address repeated prompts and better utilize the limited GPU quota, the router is coupled with a semantic cache backed by Redis [12]. For each incoming request, we first compute a fixed-dimensional embedding using a transformer-based sentence encoder, *all-MiniLM-L6-v2 model* was used. The cache stores triples of the form (embedding, normalized query text, response) as Redis entries and maintains an in-memory vector index over all cached embeddings. Instead of only caching verbatim strings, we compare the embedding of the new request against cached embeddings using cosine similarity,

$$\text{cosine\_sim}(u, v) = \frac{u \cdot v}{\|u\| \, \|v\|},$$

and treat an entry as a cache hit if the similarity score exceeds a threshold τ (in our experiments, $\tau \approx 0.95$). Each cache entry is also assigned a time-to-live (TTL) so that stale responses are eventually evicted.

Conceptually, the flow is:

1) Compute or retrieve a 768-dimensional embedding for the incoming request using the sentence encoder.
2) Compute cosine similarity between this embedding and all cached embeddings and query the vector index for the nearest neighbor above a similarity threshold τ.
3) If a match is found, return the cached answer and update cache statistics (for example, hit counters).
4) Otherwise, route the request through the normal CPU/GPU scheduling path, obtain a fresh response from the LLM, and store the new (embedding, query, answer) tuple back into Redis for future reuse.

This semantic caching strategy reduces redundant GPU calls for frequently asked or slightly rephrased questions, which we observed during testing and demos. The effect is to "stretch" limited GPU capacity further, reserving it for genuinely new or complex queries that cannot be served from cache, while keeping end-to-end latency low for repeated queries.

4. Kubernetes Deployment and Autoscaling

We deploy the system on two Kubernetes clusters: GKE on GCP and the Nautilus shared research cluster. In both environments, model backends and the router run as separate `Deployment` objects behind stable `Service` endpoints.

On GKE, we maintain separate deployments for CPU-based and GPU-based backends. Due to quota limitations, only a single GPU node (with an NVIDIA L4) is available, so the GPU backend runs with a fixed replica count. The CPU backend, in contrast, is autoscaled with a Horizontal Pod Autoscaler (HPA) based on observed CPU utilization: when load increases or many requests are routed to CPU, the HPA scales the number of CPU backend pods up, and scales them back down when traffic subsides. The router service is also placed behind an HPA, since it sits on the critical path for all requests and can become a bottleneck if it is overloaded. Figure 1 summarizes how these components are deployed together on GKE, including the router, CPU/GPU backends, Redis, and the monitoring stack.

On Nautilus, we use a single deployment of GPU-based backends only, and rely on Kubernetes scheduling preferences to target different GPU types (NVIDIA L40 as the primary choice, with Tesla V100 and L4 as fallbacks). This deployment is also autoscaled via an HPA based on CPU utilization. Although the underlying hardware differs between GKE and Nautilus, the autoscaling strategy is consistent: GPU capacity remains fixed and scarce, while CPU-backed inference and the router scale elastically with demand. The resulting topology on Nautilus is illustrated in Figure 2.

5. Frontend

The frontend is a standalone lightweight React single-page application built with Vite. It provides a chat-style interface where users can enter legal questions, view model responses, and request backend choice (such as CPU, GPU, or cache). The UI communicates exclusively with the router via a simple JSON over HTTP API, which keeps the client stateless.

6. Monitoring with Prometheus and Grafana

To make the behavior of the system observable under different workloads, we deploy Prometheus and Grafana alongside the application stack. Prometheus periodically scrapes metrics from the router, CPU and GPU backends, Redis, and Kubernetes components (nodes, pods, and HPAs) via their `/metrics` endpoints. These metrics include request latency and throughput, cache hit rates, inference times, tokens generated, pod restarts, and basic resource usage (CPU and memory). Grafana is configured with dashboards that visualize these time series, allowing us to correlate traffic patterns with backend behavior and resource utilization. During load, soak, spike, and stress tests, the dashboards make it straightforward to see when the router starts falling back to CPU, when the HPA scales additional pods, and how effectively the semantic cache reduces repeated GPU work.

C. Deployment Pipeline on GCP

On GCP, we use a container-based deployment pipeline targeting Google Kubernetes Engine (GKE). Application code for the router, CPU backend, and GPU backend is packaged into Docker images and pushed to a container registry, after which versioned Kubernetes manifests are applied to the GKE cluster to roll out updates. Model artifacts (quantized GGUF files and configuration) are stored separately in Google Cloud Storage (GCS) and pulled into the pods at startup, keeping images small and allowing model updates without rebuilding containers. Configuration such as backend URLs, resource limits, and feature flags (for example, enabling semantic cache) is injected via Kubernetes `ConfigMap` and `Secret` objects, so the same images can be reused across development and production-like namespaces. Figure 3 summarizes this pipeline, from Cloud Build to GCS-backed model artifacts and GKE rollouts.

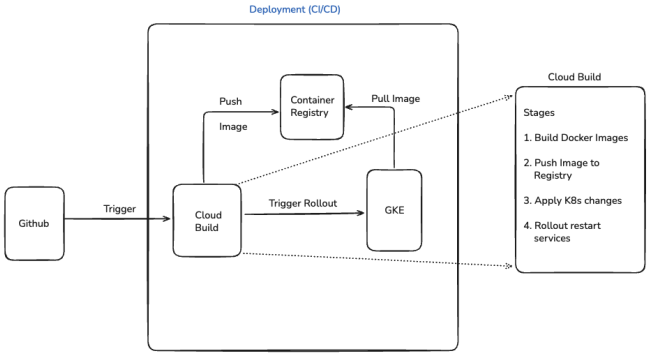

Fig. 3. Container-based deployment pipeline on GCP: Cloud Build builds and pushes images, model artifacts are stored in GCS, and versioned manifests roll out updates to GKE.

IV. Resources Needed

The project uses the following resources:

A. Hardware

- **Local development:** A developer workstation with at least 16 GB RAM and a modern CPU or laptop GPU for debugging and small-scale inference runs.
- **GCP:**
  - GKE cluster with a mix of CPU nodes and GPU nodes for the API backends.
  - Google Cloud Storage (GCS) buckets for model artifacts.
- **Nautilus:**
  - GPU nodes with atleast NVIDIA L4 accelerators and CephFS storage.

B. Software

- Python 3.10+, PyTorch and Hugging Face Transformers for LoRA training.
- `llama.cpp` and `llama-cpp-python` bindings for inference.
- FastAPI, Uvicorn, and Redis for the backend and caching.
- React + Vite for the frontend.
- Docker and Kubernetes for containerization and orchestration.
- Prometheus and Grafana for monitoring and visualization.
- k6 or equivalent load-testing tool for benchmarking.

V. Evaluation And Cost

A. Benchmark and Performance Testing Suite Overview

Evaluating LLM inference systems is non-trivial, model behavior is opaque, performance varies significantly by hardware, and end-to-end latency depends on multiple interacting microservices. To measure the efficiency of our serving pipeline, we develop a dedicated benchmark suite that isolates system-level behavior from model correctness. This suite enables controlled experimentation across different workload profiles and provides quantitative insight into latency, throughput, cache effectiveness, and system stability. All experiments were conducted in an isolated environment to avoid external interference.

B. Benchmark Directory Structure

The `benchmark/` directory (see test repository) contains all scripts and configurations used for performance testing.

- **`config/`** — Holds `params.json`, which defines all parameters for testing all scenarios. The `prompts/` subfolder provides different datasets for testing.
- **`scenarios/`** — k6 scripts implementing different workload patterns: `load`, `stress`, `spike`, `soak`, and `endurance`. Each scenario imports the shared runner and only differs in traffic shape.

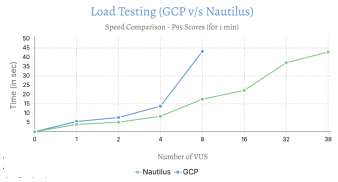

Fig. 4. GCP v/s Nautilus (P95 Scores)

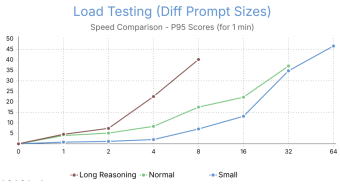

Fig. 5. Load Testing on Different Prompt Sizes (VUS v/s P95 Speed)

- **`common.js`** — Provides Abstraction for scenario testing.
- **`results/`** — Stores raw k6 outputs from each run for later analysis.
- **`modelBenchmark/`** — Uses Sentence transformer and LLM-As-Judge to evaluate model performance.

C. Benchmark Performance

To evaluate CounselGPT under realistic operational conditions, we conducted controlled load experiments using the k6 framework. Each test measured P95 latency, maximum sustainable virtual users (VUS) and failure rates under fixed conditions (failure rate $< 1\%$, 1-minute duration, 0.2-second think-time). The tests were executed in both Google Cloud (GCP) and the Nautilus research cluster to compare the behavior of different GPU environments. We terminate performance measurements once the failure rate exceeds 1%, as comparing workloads under erroneous and error-free conditions does not constitute a fair or meaningful evaluation.

1) GCP vs. Nautilus Throughput: Nautilus consistently outperformed GCP for llama.cpp workloads as seen in Figure. 4

- Nautilus maintained **1.3 RPS**, compared to **0.3 RPS** in GCP.
- The maximum stable concurrency was **38 VUS** vs. **8 VUS** on GCP.

This indicates that Nautilus' V100/L40 GPUs provide significantly more stable inference throughput than the smaller GCP GPU used in our deployment .

2) Impact of Prompt Size: Prompt is the main metric affecting how latency works in LLM's as activation functions depend on it. This is clearly visulized in our load testing with similar load on various prompts as depicted in Figure. 5.

- The small prompts reached **2.4 RPS**.
- Normal prompts averaged **1.3 RPS**.
- Long reasoning prompts dropped to **0.3 RPS**.

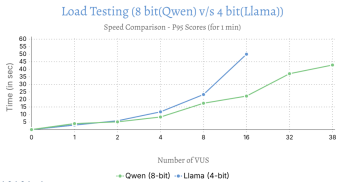

Fig. 6. Qwen (8-bit) v/s Llama (4-bit) P95 scores

Large Reasoning prompts takes more time and have more errors rate when increases no of virtual users more than 8 on the current system, while small and normal prompts perform better. Larger prompts increase KV-cache operations and attention computations, leading to near-linear latency growth.

3) Qwen (8-bit) vs. Llama-2 (4-bit): The normal understanding is that 4-bit should be even more faster than 8-bit because of the number of bits, but our tests and research clearly proved it is not necessarily has to be same. Despite being higher precision, Qwen2.5-7B (8-bit) performed better than Llama-2-7B (4-bit) as seen in Figure. 6. Resource utilization effect on using different bit models is reported in the *System Performance and Sustainability section*.

- Llama sustained only **0.5 RPS**, while Qwen achieved **1.3 RPS**.
- Qwen's architecture (efficient attention, fewer layers) reduces FLOPs/token.

This demonstrates Qwen's architectural advantage for latency-sensitive reasoning tasks.

4) Stability Across Test Modalities: Below table depicts all benchmarks performed against our best model (Qwen) in our performative cloud (Nautilus).

Test	Max VUS	P95 Latency (sec)
Load	4	8.09
Stress	25	41.47 *
Spike	25	29.39
Soak (10 min)	15	39.73
Endurance (2 hr)	15	36.20*

TABLE I
SUSTAINED CONCURRENCY AND P95 LATENCY ACROSS BENCHMARK TYPES

Across soak and endurance tests, the system remained stable with moderate latency degradation. This confirms that llama.cpp-based serving can handle multi-minute and multi-hour workloads reliably on Nautilus GPUs. Stress tests takes more than error rate and performs the output, which could be due to the system not being handle increased loads when there is already a lot of requests in pipeline. (* - Error rate observed more than 1%)

D. Semantic Caching Analysis

To study the impact of semantic caching on inference efficiency, we issued a set of legally similar prompts shown in Figure. 7 designed to trigger high embedding similarity.

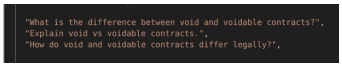

Fig. 7. similar.json - Similar prompts to test Semantic caching

Fig. 8. Cache hits v/s Tokens Generated. (Top Pre-cache and Bottom Post-Cache)

A 95% cosine-similarity threshold was used to qualify cache hits.

Figure. 8 compares pre-cache and post-cache behavior. Before caching, the model generated full responses for every request, producing full tokens/sec and zero cache hits. Once the cache warmed up, cache hits increased and the model stopped generating tokens for repeated or near-duplicate queries. This led to a dramatic less in tokens/sec (as the GPU no longer computed full forward passes), and cache hit rates gradually overtook misses.

Our test results shows that Semantic caching reduces around **32%** of tokens generated when compared with the same set of prompts without semantic caching at 95% threshold. Semantic caching reduces GPU utilization because repeated questions are served directly from Redis without running the LLM. This lowers inference cost, improves sustainability, and increases request throughput. However, as this mechanism relies on embedding similarity instead of exact key matches, more evaluation is required to determine its effectiveness in real production workloads.

Future work includes testing cache usefulness in user-driven, real-world traffic, studying threshold sensitivity, and understanding whether semantic caching continues to provide benefit when prompts vary widely.

E. Additional Evaluation Factors

In addition to load- and stress-based benchmarking, we evaluated two further dimensions of the CounselGPT system: (a) hardware efficiency and (b) model quality. These tests provide complementary insight into how the system behaves under different resource constraints and accuracy requirements.

Hardware Efficiency: CPU vs. GPU Throughput: To quantify the benefit of hardware acceleration, we executed identical workloads on CPU-only and GPU-backed configurations over a five-minute window. The GPU-enabled setup delivered a dramatic throughput increase, highlighting the importance of accelerator hardware for serving quantized LLMs at scale.

Model Quality Comparison: We also assessed output quality across the two deployed models , Llama-2 (4-bit) and Qwen-2.5 (8-bit). Using Sentence Transformer similarity scoring and LLM-as-Judge for verification, Qwen demonstrated stronger alignment with ground-truth answers despite higher quantization, suggesting architectural efficiency advantages.

Test	Throughput (5 min)
Without GPU Acceleration	1
With GPU Acceleration	52

TABLE II
THROUGHPUT COMPARISON BETWEEN CPU-ONLY AND GPU-BACKED INFERENCE.

Model	Performance Score
Llama-2 (4-Bit)	75%
Qwen-2.5 (8-Bit)	90%

TABLE III
RELATIVE MODEL ACCURACY MEASURED USING LLM-AS-JUDGE AND SIMILARITY METRICS.

Together, these tests illustrate that system performance is shaped not only by serving architecture and load characteristics but also by hardware selection and model design choices.

F. Metrics and Visualization

To understand the runtime behavior of CounselGPT under varying load patterns, we instrument the system with a lightweight metrics pipeline built on Prometheus and Grafana. Prometheus periodically scrapes numerical counters and histograms exported by the API, LLM backend, cache, and system runtime. Grafana renders these signals into interactive time-series panels for real-time debugging and post-benchmark analysis.

Metrics Captured: The dashboard aggregates all metrics relevant to LLM serving:

- **Inference performance:** Average latency, P50/P90/P95/P99 histograms, tokens generated per second, and total tokens generated over a window.
- **Traffic and throughput:** HTTP requests per second, average tokens per request, and error rates.
- **Cache efficiency:** Hit/miss rate, hit ratio over sliding windows, and cache operations per second.
- **System resource and runtime health:** CPU usage per pod, memory RSS, virtual memory usage, open file descriptors, process-level memory footprint, and basic service liveness/readiness gauges.
- **GPU utilization:**
  - GCP: collected using the NVIDIA `dcgm-exporter`.
  - Nautilus: GPU utilization is provided by the platform itself, as DCGM cannot be deployed due to security restrictions.
- **Router metrics:** End-to-end request latency by backend (CPU, GPU, or cache), distribution of traffic across backends, counts and reasons for fallbacks, and circuit-breaker state over time (e.g., fraction of time the GPU path is open vs. half-open or open).

These metrics form the basis for understanding bottlenecks such as small-batch token throughput, cache behavior, memory fragmentation, and pod-level performance variance.

Fig. 9. L40 v/s V100 v/s L4 GPU Utilization comparison

Fig. 10. GPU System Metrics

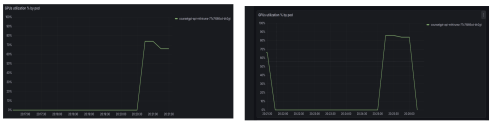

Fig. 11. GPU Utilization 8-bit(left) v/s 4-bit(right)

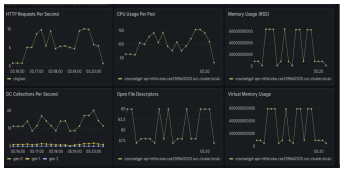

Fig. 12. System Metrics

Key Computation Formulas: We rely on a small set of PromQL expressions to derive actionable performance indicators:

- **Average inference latency (5m window):**

$$\frac{\text{rate}(duration\_sum[5m])}{\max(\text{rate}(duration\_count[5m]),\, 1)}$$

- **Tokens per second:**

$$\text{sum}\big(\text{rate}(tokens\_generated\_total[1m])\big)$$

- **Cache hit ratio (5m):**

$$\frac{\text{increase}(hits[5m])}{\text{increase}(hits[5m]) + \text{increase}(misses[5m])}$$

Grafana Dashboard Layout: Our Grafana dashboard organizes the collected metrics into:

- LLM latency and quantile trends;
- Token-generation and throughput patterns;
- Cache behavior panels (hit ratio, hits vs. misses);
- CPU, memory, Speed (P-50 and all tail latencies) and GC-related runtime performance;
- GPU utilization (GCP DCGM / Nautilus native exporter).

This visualization layer provides an end-to-end operational view of CounselGPT, allowing correlation of traffic spikes with resource saturation, cache efficiency, and inference slowdowns. Grafana dashboard for both cloud can be viewed here.[6], [7], [8]

G. System Performance and Sustainability

Across all benchmark scenarios, the Nautilus cluster consistently sustained more than 20 concurrent VUS regardless of prompt type. Our P95 scores were well within 40 seconds for 40 VUS and P50 scores were around 20 seconds showing efficiency. This is primarily due to access to higher–bandwidth data-center GPUs such as the NVIDIA L40 and V100, whereas GCP tests were executed on the lighter L4 GPU. Despite lower FP16 TFLOPs, the L4 remains a cost–efficient option because it can be horizontally scaled and achieve good performance. When speaking about GPU Utilization V100 and L40 operated at substantially lower percentages for similar workloads when compared with L4 when tested on Nautilus as seen in Figure. 9. This difference arises from architectural throughput: the L40 and V100 provide significantly higher memory bandwidth and compute parallelism, allowing them to serve tokens without saturating the device.

The NVIDIA L4 showed strong energy-efficient behavior during the spike test, processing a sudden jump to 25 users while drawing only around 70 watts as observed in GCP 10. Its power usage stayed stable instead of rising sharply, meaning each request consumed less energy and produced a lower carbon footprint. The GPU also held a safe temperature range and used memory consistently, avoiding waste and keeping cooling overhead low. Overall, the L4 delivered high throughput per watt, making it a more sustainable choice for running LLM inference workloads.

One more interesting observation is that 4 bit takes more utilization than 8 bit as seen in Figure. 11. Our tests showed the 4-bit model reaching 85–90% GPU utilization, while the 8-bit model remained around 65–70%, not because 4-bit is heavier, but because it fully saturates the GPU's compute capacity. The main reason we believe is lazy loading of Llama 4-bit model, which can cause the utilization to be more.

System-level metrics (RAM usage, open file descriptors, GC activity, and virtual memory) shown in Figure. 12 stable behavior across tests, indicating no resource leaks during long-running workloads. GPU telemetry on the L4 (utilization, temperature, power draw) further confirmed thermal stability and predictable performance under sustained inference.

Overall, the system demonstrates good operational sustainability: Nautilus offers high concurrency at zero cost using stronger GPUs.

H. Cost Analysis

We used approximately $ 40 of our GCP credits allocated for this project. These credits were used to benchmark our system on an NVIDIA L4 GPU instance. Although Nautilus provides GPUs at no monetary cost, we use GCP's Tesla V100 hardware cost to compare side-by-side to give us a realistic

reference point to understand how much we *would* pay for similar performance in the cloud.

Under identical load-testing conditions, we observed:

- **GCP L4:** p95 latency $\approx$ 19 seconds, throughput $\approx$ 0.34 req/s
- **Nautilus V100:** p95 latency $\approx$ 8.09 seconds, throughput $\approx$ 0.64 req/s

Since Nautilus is free, we computed cost-efficiency metrics only using the GCP L4 and V100 pricing (on-demand). Using the measured throughput, we derive cost-normalized metrics. We additionally compute the estimated *cost per 1000 requests*:

Platform	RPS	Cost
GCP L4	0.34	$0.71
Nautilus V100	0.64	$1.77

TABLE IV
COST AND RPS COMPARISON.

The comparison shows that although the Nautilus V100 incurs a higher nominal hourly cost when priced using cloud-equivalent rates, it delivers nearly double the throughput of the GCP L4 instance (0.64 RPS vs. 0.34 RPS). This demonstrates that V100-class hardware provides significantly better performance per unit time for our workload. Since Nautilus resources are free to use, the effective cost-performance advantage becomes even more pronounced, making it the most economical and scalable platform for our inference pipeline.

VI. Conclusion and Future Work

Conclusion. Through our experiments, we observed that system performance in LLM inference is strongly tied to GPU capability rather than quantization alone; in particular, 8-bit Qwen often outperformed 4-bit Llama despite higher precision, highlighting that model architecture can outweigh numeric compression. Nautilus nodes, equipped with L40 and V100 GPUs, handled significantly higher concurrency than GCP's L4, confirming that hardware class directly governs throughput. We also found that cold starts introduce substantial latency when new replicas load multi-gigabyte weights, making horizontal pod autoscaling effective for availability but unsuitable for real-time scaling. Although CPU fallback improves resilience, it is not practical for inference workloads once traffic exceeds trivial levels.

Future Work. Several directions remain to be explored. A like-for-like GPU comparison was not feasible due to resource constraints, which limited the fairness of cross-cloud evaluation. Future work includes obtaining equivalent GPU models across both platforms to enable a more balanced performance comparison. Semantic caching showed promising reductions in token generation and GPU load, but requires deeper evaluation across diverse workloads and similarity distributions. GPU-aware scaling strategies, batching, and model-pool routing could further improve throughput. Additionally, while llama.cpp is efficient on lightweight or edge devices, data-center GPUs demand serving systems that minimize KV-cache fragmentation and maximize memory reuse. A natural extension is migrating to `vLLM`, whose paged-attention architecture achieves near-zero KV-cache waste and significantly higher throughput [5]. Integrating such backends, along with broader benchmark coverage, forms the next step toward a fully optimized and sustainable legal-domain LLM serving system.

All our live deployment link, codebase and documentation is present in these links. [9], [10], [11]

VII. Peer Deployment Excercise

Our group was evaluated by Haardhik and is peer deployment review is attached at the end of this document. We would like to thank Haardhik to take time and deploying our model to Nautilus. We would also like to thank Professor Abel Souza for giving us $ 50 credits and all access to Nautilus cluster to actually learn more about Inference optimization.

References

[1] E. J. Hu *et al.*, "LoRA: Low-Rank Adaptation of Large Language Models," in *Proc. ICLR*, 2022.

[2] G. Gerganov *et al.*, "llama.cpp: LLM inference in C/C++," GitHub repository, 2023. [Online]. Available: https://github.com/ggml-org/llama.cpp

[3] T. Wolf *et al.*, "Transformers: State-of-the-Art Natural Language Processing," in *Proc. EMNLP*, 2020.

[4] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou, "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models," in *Proc. NeurIPS*, 2022.

[5] W. Kwon, Z. Li, S. Zhang, Y. Zheng, L. Zheng, D. Papailiopoulos, J. E. Gonzalez, and I. Stoica, "Efficient Memory Management for Large Language Model Serving with PagedAttention," in *Proc. ACM Symp. Cloud Comput. (SoCC)*, 2023.

[6] Nautilus GPU Dashboard, Grafana, 2025. [Online]. Available: https://grafana.nrp-nautilus.io/d/dRG9q0Ymz/k8s-compute-resources-namespace-gpus?var-namespace=cse239fall2025&orgId=1&from=now-5m&to=now&timezone=browser&refresh=1d&var-Filters=pod%7C%21%3D%7Cllm-inference-gpu-6454d768c4-wmqxs&var-Filters=pod%7C%21%3D%7Cllm-inference-768dd8c5cf-txnnj

[7] CounselGPT LLM Metrics Dashboard, Grafana, 2025. [Online]. Available: https://grafana-mathesh.nrp-nautilus.io/d/cf69ukbzmuw3kf/counselgpt-clean-llm-metrics-dashboard

[8] GCP Grafana Monitoring Dashboards, 2025. [Online]. Available: https://34.111.194.27.nip.io/grafana/dashboards

[9] CounselGPT Online, 2025. [Online]. Available: https://vasavisd.github.io/CounselGPT/

[10] CounselGPT API and deployment code, 2025. [Online]. Available: https://github.com/Mati02K/CounselGPT/tree/main

[11] CounselGPT - LoRA Training Code, 2025. [Online]. Available: https://github.com/VasaviSD/lora-legal-training

[12] Redis - Semantic Caching. [Blog] https://redis.io/blog/what-is-semantic-caching/