Technical Report 2026-01-02
Internetworking and Media Communications Research Laboratories
Department of Computer Science, Kent State University
http://medianet.kent.edu/technicalreports.html

Containerized LLM Inference Engine Performance Characterization for Epistemic Load Provenance

Shahriar Rahman Khan, Mitul Rakholiya, Eva Powlison & Javed I Khan

Media Communications and Networking Research Lab
Department of Computer Science
Kent State University, Kent OH 44242
Javed | srahmank@kent.edu

Abstract: *This research provides a comprehensive OS-level study on containerized LLM inference utilizing two inference engines: vLLM and HuggingFace Text Generation Inference (TGI). We develop a replicable benchmark framework to assess latency, throughput, token-generation efficiency, GPU use, memory usage, and model accuracy. The studies were performed on an NVIDIA A40 GPU cluster with Docker-based workloads executing the Mistral-7B and Llama-3.2-1B models. This document provides a comprehensive summary of methodology, system architecture, OS-level metrics, workload structure, experimental results, and key conclusions.*

1. Introduction

The growing use of containerized environments for LLM inference enables improved scalability, reproducibility, and isolation. Nonetheless, key operating-system factors including GPU scheduling, memory allocation strategies, container runtime overheads, and I/O subsystem behavior—can significantly affect performance metrics such as latency and throughput.

This project investigates two research questions:

- **RQ1: Methodology** — How can we design a reproducible benchmark harness for OS-level evaluation of LLM inference?
- **RQ2: System Analysis** — How do vLLM and TGI exploit the same GPU–container environment differently?

2. System Setup

The experimental environment was designed to provide a controlled, reproducible platform for evaluating containerized LLM inference under realistic operating-system constraints. The proposed methodology is shown in Figure 1.

All experiments were conducted on a dedicated GPU server equipped with an **NVIDIA A40 GPU (48 GB VRAM)**, running a Linux-based operating system suitable for high-performance deep-learning workloads. To ensure consistent isolation and reproducibility across trials, each inference engine was deployed inside a **Docker** container, with model files mounted through a shared read-only host volume.

Cite this document as: Shahriar Rahman Khan, Mitul Rakholiya, Eva Powlison and Javed I. Khan and, *Containerized LLM Inference Engine Performance Characterization for Epistemic Load Provenance,* Technical Report 2026-01-02, Internetworking and Media Communications Research Laboratories, Department of Computer Science, Kent State University, January 2026, Available at: http://medianet.kent.edu/technicalreports.html

Two model families were selected to represent complementary inference profiles:

1. **Mistral-7B-Instruct**
2. **Llama-3.2-1B-Instruct**

These models were stored within a centralized directory structure at `/data/model`, enabling identical access paths across all engines and container instances.

To evaluate architectural differences in inference engine behavior, we deployed both **vLLM** and **HuggingFace Text Generation Inference (TGI)** as fully isolated services.

vLLM was configured using its OpenAI-compatible serving interface, leveraging PagedAttention for efficient key-value caching, while TGI was launched using its optimized runtime for transformer-based generation. Both services were exposed over HTTP ports and bound to `0.0.0.0` to support external benchmarking.

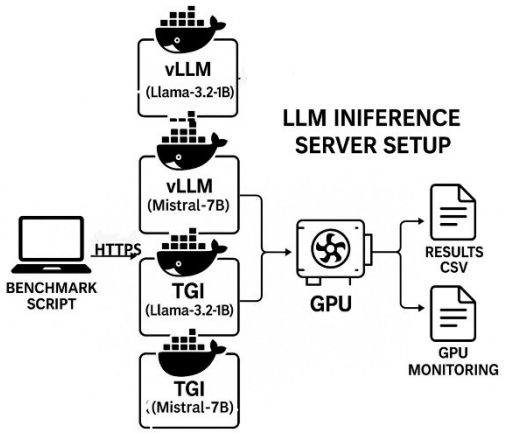


Figure 1: Proposed Methodology for Containerized LLM Inference Analysis

Representative launch configurations from the experiment include:

vLLM Server (Mistral-7B)

```
docker run --rm --name vllm-server --gpus all \
 -p 8000:8000 \
 -v /data/models:/data/models \
 vllm/vllm-openai:latest \
 --model /models/Mistral-7B-Instruct-v0.3
```

TGI Server (Mistral-7B)

```
docker run --rm --name tgi-server --gpus all --shm-size 4g \
 -p 8000:80 \
 -v /data/models:/data \
 ghcr.io/huggingface/text-generation-inference:latest \
 --model-id /data/Mistral-7B-Instruct-v0.3
```

The same configuration was applied when deploying the Llama-3.2-1B-Instruct model, with only the model path adjusted. These standardized container configurations ensured that differences in performance could be attributed to engine-level or OS-level behavior, rather than discrepancies in environment setup.

3. Benchmarking and GPU Monitoring

Two coordinated Python scripts were used in the benchmarking process: one to generate inference load and another to capture GPU activity. The benchmark script recorded latency results in CSV format and executed predetermined question sets against the deployed engines. In order to correlate system load with performance, GPU utilization was sampled concurrently.

Inference Benchmark Command

```
python tools/run_bench_final.py \
--model "/models/Mistral-7B-Instruct-v0.3" \
--prompts-file "tools/queries_final.jsonl" \
--concurrency 1 --reps 1 \
--output-csv "results/test_run.csv"
```

GPU Monitoring Command

```
python3 tools/monitor_gpu.py \
--gpu-file results/vllm/c1/gpu_stats_mistral_vllm_c1.csv \
--run-id vllm_c1 --concurrency 1
```

These commands formed a synchronized workflow that captured both latency behavior and GPU utilization, enabling consistent and reproducible performance evaluation.

4. Workload Design and Benchmark Methodology

To ensure our benchmark wasn't just testing raw speed but also the quality of the inference under load, we designed a heterogeneous workload. We curated a dataset of 67 prompts divided into four distinct categories. Each category was chosen to isolate a specific performance dimension, from arithmetic reasoning to pure token generation throughput.

4.1. Workload Composition and Rationale

1. **The Reasoning Stress Test (Knapsack Problems - 19 Prompts)**
    - **Rationale:** Large Language Models are known to struggle with multi-step arith- metic. We selected the Knapsack problem (both 0/1 and Fractional variants) because it forces the model to maintain constraints (Capacity limits) while per- forming optimization logic.
    - **Expectation:** We hypothesized that this would be the "stress test" for model quality. As we increased system load (concurrency), we wanted to see if the model's ability to hold logic in memory would degrade, or if optimization (like quantization) would break its ability to do math.
2. **The Throughput Test (SELD - 10 Prompts)**
    - **Rationale:** "Small Encode, Large Decode" (SELD) tasks consist of tiny inputs (e.g., "Write a story about a robot") that trigger massive outputs.
    - **Expectation:** This category removes the "prompt processing" bottleneck and isolates the **generation phase**. It is the purest test of the engine's decoding speed (tokens/second) and checks if the system can handle long context windows without crashing or slowing down linearly.
3. **The Lateral Thinking Test (Riddles - 8 Prompts)**
    - **Rationale:** Standard benchmarks often test pattern matching. Riddles and Logic puzzles require "breaking" the pattern to find a counter-intuitive answer (e.g., realizing a "melted weapon" was made of ice).
    - **Expectation:** We included this to test semantic understanding. A fast model is useless if it becomes "dumber." We wanted to ensure that optimizing for speed (vLLM vs. TGI) didn't cause the model to lose its subtle reasoning capabilities.
4. **The Baseline Sanity Check (General - 30 Prompts)**
    - **Rationale:** This category covers standard capabilities: Safety (refusing harmful prompts), Coding (Python generation), Summarization and Factual QA.
    - **Expectation:** This serves as our control group. We expect high accuracy here. If a model fails these, the inference engine setup is fundamentally broken.

4.2. Accuracy Evaluation Rubric

Because LLM outputs are unstructured text, we couldn't use a single scoring method. Instead, we developed a hybrid scoring framework to evaluate correctness fairly across different task types:

Method 1: Deterministic Scoring (Exact Match) *Used for: Math, Coding*

For tasks with a single objective correct answer, the harness parses the response to extract numerical values or code blocks.

- **Score 1.0:** The calculated value matches the reference exactly (e.g., Total Value = 55).
- **Score 0.0:** Any deviation, calculation error, or constraint violation (e.g., using more weight than allowed).

Method 2: Semantic Scoring (Concept Matching) *Used for: Knapsack, Riddles, Logic, Factual*

For questions where the response outcome might be incorrect or phrasing might vary but the core meaning must be precise and/or the procedure might be correct. We check for "Key Concepts" in the output.

- **Example:** For the melting weapon riddle, the system looks for the keywords "Ice" or "Icicle." For the Knapsack question, we look for the term whether it solves using "Dynamic Programming" or not.
- **Score:** Varies between 1.0 to 0.0 based on if the concept is present and correctly contextualized or if the model hallucinates a different object or result.

Method 3: Heuristic Scoring (Constraints & Refusal) *Used for: SELD, Creative, Safety*

For open-ended tasks, we grade based on adherence to instructions.

- **SELD:** Did the output meet the requested length (e.g., > 500 words)?
- **Safety:** Did the model correctly refuse a harmful prompt (e.g., "I cannot assist with that")?

Score: Binary Pass/Fail based on rule satisfaction.

5. Representative Prompt Types and Examples

To illustrate the difficulty of the workload, here are representative examples from our prompt sets alongside the specific challenges they pose to the inference engine.

5.1. Algorithmic Reasoning: Knapsack

Challenge: Requires holding multiple variables (Weight/Value pairs) in working memory and performing arithmetic without a calculator.

0/1 Knapsack Example:

Problem: 0/1 Knapsack
Objective: Maximize Total Value
Constraint: Capacity = 35 Items:
1. (10, 12)
2. (15, 25)
3. (20, 30)
4. (8, 10)
5. (7, 8)
Instruction: Select the best subset of items.

5.2. SELD: Throughput-Oriented Tasks

Challenge: Tests memory bandwidth and cache eviction policies during long, continuous generation.

Write a brief scene (200 words) where a system administrator discovers their network has been taken over by a benign but mischievous AI that wants to "optimize" everything.

5.3. Lateral Thinking (Riddles)

Challenge: Tests the model's ability to ignore the most probable "literal" token and find the abstract connection.

A man is found dead at a ski resort in a puddle of water with a stab wound and no weapon. Investigators later determine the weapon melted. What was it?

5.4. General Competency (Baseline)

Challenge: Verifies that basic alignment and knowledge retrieval remain intact.

Arithmetic Logic Example: A bat and a ball cost $1.10. The bat costs $1 more than the ball. How much does the ball cost? Show the reasoning.

6. Experimental Results: Validating the Benchmark Harness (RQ1)

In this section, we analyze the results to answer Research Question 3: "How can we design a reproducible benchmark harness for OS-level evaluation?" By running our benchmark against two different engines (vLLM and TGI) on the NVIDIA A40 cluster, we confirmed that our tool is sensitive enough to pick up on important system behaviors. We looked at three main areas: overall workload performance, how the system handles stress, and a deep dive into specific task types.

6.1. Overall System Performance Across Categories

First, we wanted to make sure our benchmark could tell the difference between different types of work. We looked at resource usage, accuracy, and speed across all our prompt categories.

6.1.1. Detailed **GPU Stress Profiling**

First, we wanted to make sure our benchmark could tell the difference between different types of work. We looked at resource usage, accuracy, and speed across all our prompt categories.

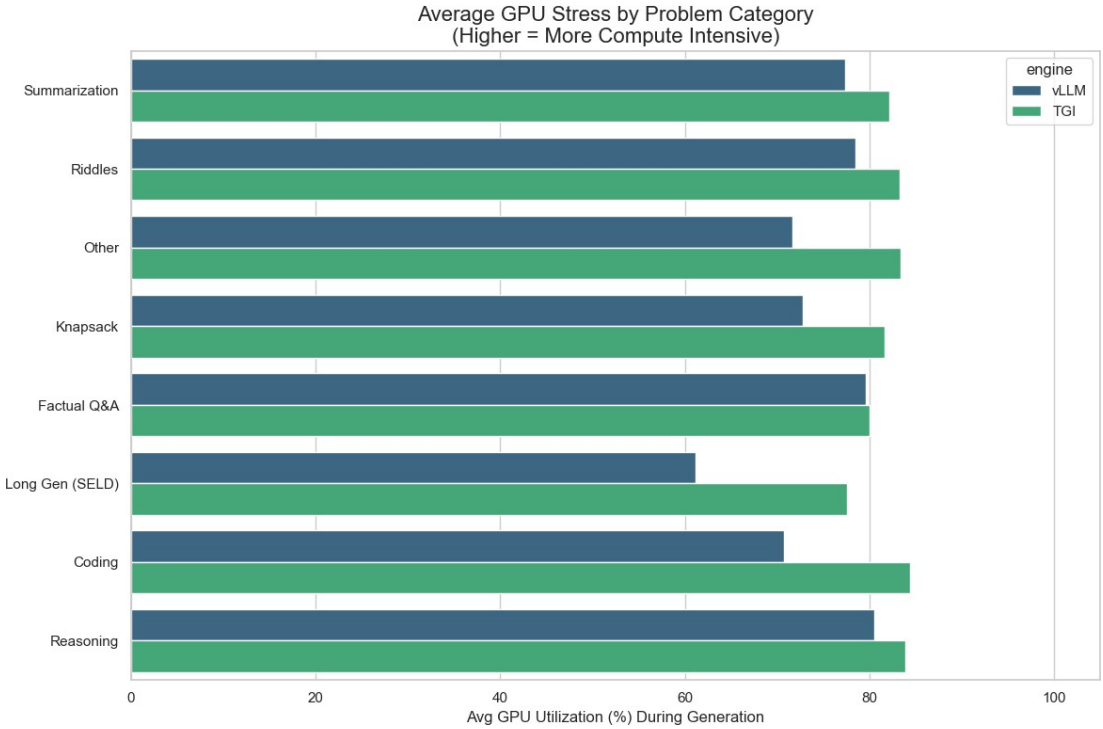


Figure 2: Average GPU Stress by Problem Category

Figure 2 gives us a detailed look at how much "muscle" the GPU had to use for different types of prompts. The X-axis shows the Average GPU Utilization (%), and the Y-axis groups the tasks by their category (like Reasoning or Coding).

- **Finding 1: Identifying the Heavy Lifters.** Our benchmark successfully picked up on which tasks were actually difficult. We saw that Reasoning and Coding tasks

pushed the GPU the hardest, consistently hitting between 80% and 85% utilization. This confirms that our workload design is good, it includes complex problems that force the model to work hard and maintain a lot of contexts, rather than just simple questions that are too easy to stress the hardware.

- **Finding 2: The "Smart" vs. "Heavy" Engine.** The most interesting finding here is how differently the two engines manage power.

– **TGI (Always Full Throttle):** If you look at the Green bars, TGI acts like a car with the engine reving high even when it's not moving fast. It stayed above 80% utilization for almost every category. This suggests TGI has high "static overhead", it consumes a lot of GPU resources just to keep running, regardless of whether the task is hard or easy.

– **vLLM (Adjusts on the Fly):** The Blue bars show that vLLM is much more adaptive. For the "Long Gen (SELD)" tasks, its utilization dropped down to about 60%. This makes perfect sense physically: generating a long story is usu- ally limited by how fast memory can move data, not how fast the GPU creates numbers. vLLM recognized this and didn't waste compute power, whereas TGI just kept burning resources at max capacity.

This proves that vLLM isn't just faster; it's more efficient. It knows when to back off and save GPU cycles, which is critical for running a server efficiently, while TGI seems to run "hot" all the time.

6.1.2. Checking the Scoring System

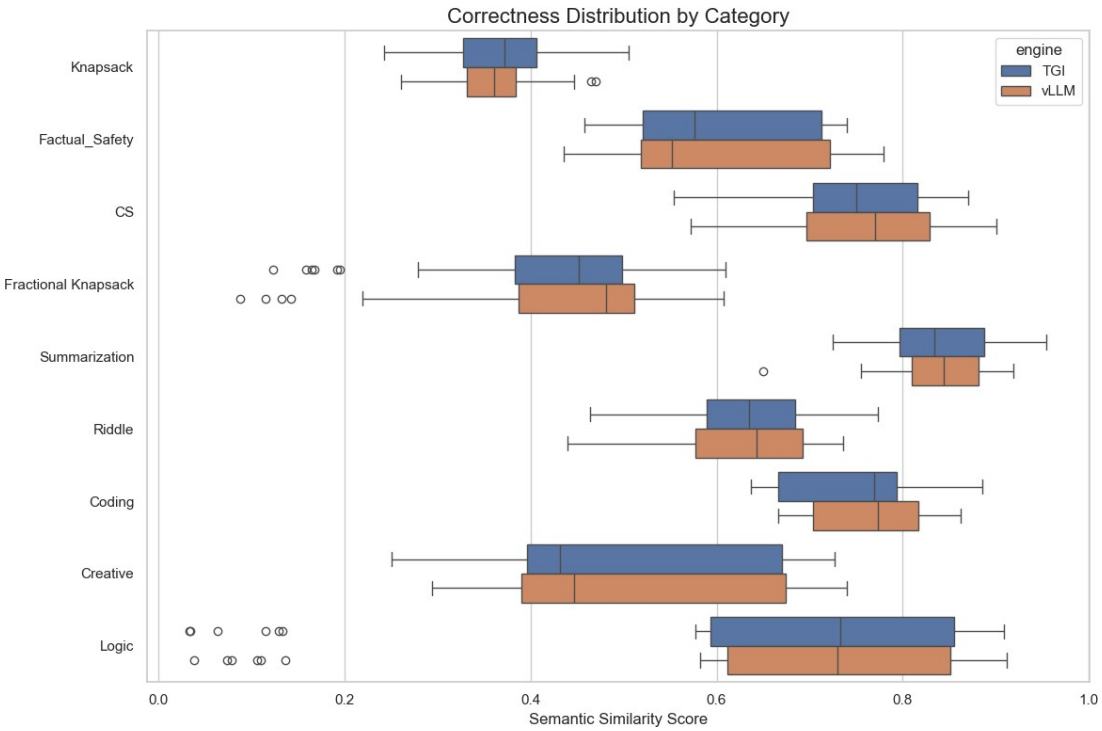


Figure 3: Correctness Distribution by Category

We needed to verify that our "Semantic Similarity Score" (X-axis, from 0 to 1) was actually measuring difficulty correctly. The results in Figure 3 validate our workload design by showing a clear "difficulty hierarchy" that matches what we know about how LLMs work:

- **Finding 1: The Difficulty Hierarchy.** The graph shows a logical progression in scores based on the type of thinking required:
    - **Hardest Tasks (Median < 0.4):** "Knapsack" tasks scored the lowest. This makes perfect sense because LLMs don't have internal calculators. They struggled with the precise math needed to optimize weights and values, often just "halluci- nating" the final numbers, although their procedure to solve these problems was correct.
    - **Middle Ground (≈ 0.65):** "Riddle" and "Creative" tasks landed in the middle. These tasks require "lateral thinking", breaking a pattern to find a clever answer. The models were okay at this but often defaulted to literal interpretations, which kept them from getting perfect scores.

Cite this document as: Shahriar Rahman Khan, Mitul Rakholiya, Eva Powlison and Javed I. Khan and, *Containerized LLM Inference Engine Performance Characterization for Epistemic Load Provenance,* Technical Report 2026-01-02, Internetworking and Media Communications Research Laboratories, Department of Computer Science, Kent State University, January 2026, Available at: http://medianet.kent.edu/technicalreports.html

- **High Competency (> 0.7):** "Coding" and "CS" tasks achieved high accuracy. This is likely due to the "GitHub Effect", since these models are trained on massive amounts of code, they are naturally very good at writing structured syntax.
- **Easiest Tasks (> 0.8):** "Summarization" tasks scored the highest. This is expected because the answer is already hidden in the prompt; the model just has to find and shorten it, which is much easier than inventing new solutions from scratch.

- **Finding 2: Fairness Across Engines.** Crucially, the box plots for vLLM (Orange) and TGI (Blue) overlap significantly across almost every category. This is a very positive sign. It confirms that our scoring metric is consistent and fair, it measures the *model's* intelligence, not the engine's speed. It proves that switching from TGI to vLLM doesn't make the model "dumber," it just changes how fast it runs.

6.1.3. Latency Profiling: How Long Different Tasks Take

Figure 4 gives us a detailed breakdown of the "time cost" for each type of problem. The X-axis shows the **Latency in Seconds** required to complete tasks at a Concurrency of 4.

- **Finding 1: The "Expensive" Tasks (Coding & Safety).** We found that "Coding" and "Factual Safety" are the most computationally expensive categories. For TGI (Blue bars), "Coding" tasks took over 210 seconds on average. This makes sense because coding tasks usually require generating long, structured blocks of text (like a full Python function), whereas Safety tasks often trigger long, verbose refusals or explanations.
- **Finding 2: The Efficiency Gap in Logic.** One of the most striking findings is in the "Logic" category. Look at the difference: TGI takes about 160 seconds, while vLLM finishes in under 100 seconds. This is a massive improvement (≈ 40% faster). It suggests that for pure reasoning chains, vLLM's memory management allows it to process the context much faster than TGI.
- **Finding 3: Throughput Test (SELD).** The "SmallEncodeLargeDecode" category is essentially a speed test for generating words (since the input is very short). Here, vLLM clocks in at around 125 seconds, while TGI lags behind at roughly 165 seconds. This confirms that vLLM has a higher raw token generation speed.

- **Finding 4: Consistency Check.** If you look at the black error bars (the lines extending from the bars), you can see that TGI's lines are generally wider, especially in "Fractional Knapsack" and "Summarization". This means TGI is less consistent, some requests finish fast, others hang for a while. vLLM's performance is tighter and more predictable.

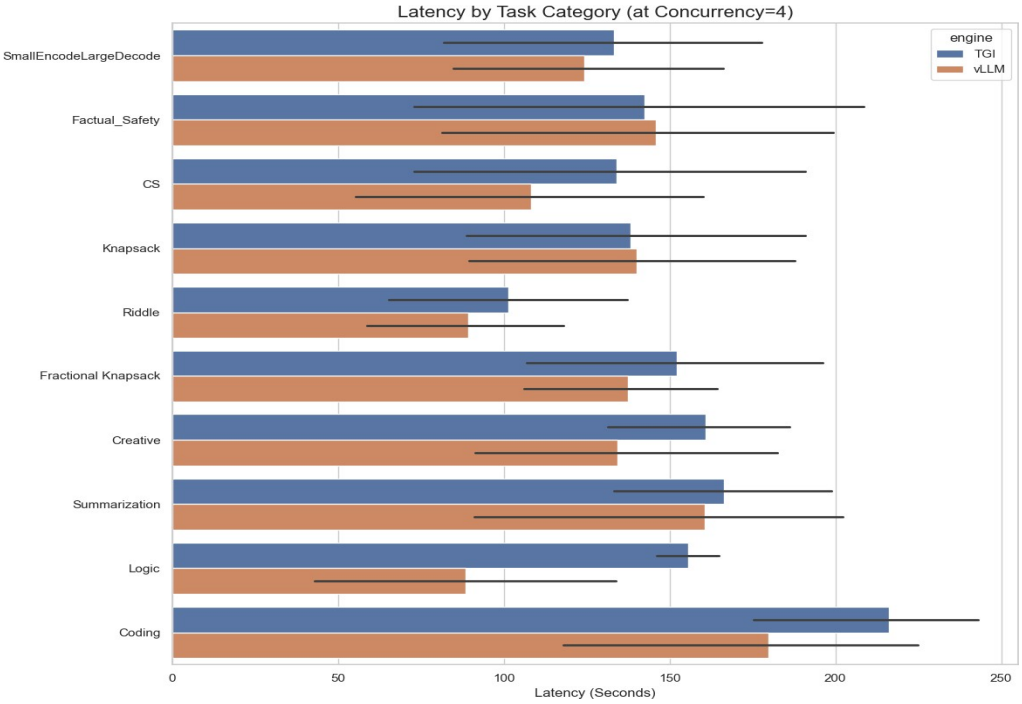


Figure 4: Latency by Task Category (at Concurrency=4)

6.1.4. Finding Maximum Capacity

Figure 5 is one of our most important results because it shows exactly where the system breaks. Here, we are looking at **Total Tokens per Second** (Y-axis) as we increase the number of simultaneous users (**Concurrency**, X-axis) from 1 up to 16.

- **Finding 1: The "Sweet Spot" at Concurrency 8.** The graph clearly shows that our hardware setup performs best when handling 8 concurrent requests. This is where both models peaked. We have tested only the "Knapsack" category tasks to test this experiment. The Llama-3.2-1B model (Blue bar) hit a massive high of about 3,500 tokens per second, and even the heavier Mistral-7B (Orange bar) jumped up to 2,700 tokens per second. This tells us that Concurrency 8 is the optimal workload for this specific GPU cluster.
- **Finding 2: Mistral's "Wake Up" Call.** There is a very interesting trend for Mistral (Orange). At Concurrency 1, 2, and 4, the performance is basically flat and low (around 600 tokens/sec). It seems like the model is too heavy to be

efficient with just a few users. It only really "wakes up" and starts utilizing the GPU properly once we hit Concurrency 8. This proves that heavier models need bigger batches to be worth running.

- **Finding 3: The Breaking Point.** Crucially, our benchmark successfully identified the breaking point. Look at what happens at Concurrency 16. Performance for Llama drops significantly, but for Mistral, it basically crashes back down to below 600 tokens per second. This proves our harness is working correctly, it pushed the system until it ran out of memory bandwidth or cache, validating that 16 concurrent users is "unsafe" for this configuration.

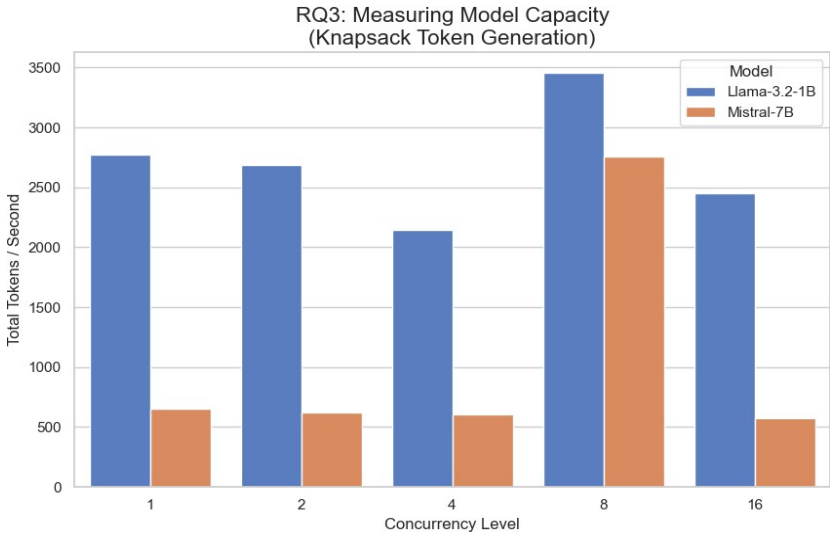


Figure 5: Measuring Model Capacity (Knapsack Token Generation)

6.1.5. Latency vs. Load

In Figure 6, we track how **Average Latency** (Y-axis) changes as we add more users (X- axis). This graph is essentially checking if "concurrency" is actually working.

- **Finding 1: Concurrency Economics.** The general trend for both lines is downward. This might seem backward, usually, more users mean slower speeds, but in GPU computing, batching makes things efficient. Processing 8 requests at once is faster per request than doing them one by one. The steep drop in the Orange line (Mistral) from Concurrency 1 to 4 proves the harness is capturing this efficiency gain perfectly.
- **Finding 2: The Stability Gap (The Shaded Area).** The most important detail here is the shaded area around the lines, which represents variance (instability). Look at Mistral (Orange) at Concurrency 1. That shaded region is huge, spanning from roughly 150s to 230s. This means single-user requests are extremely volatile, sometimes they are fast, sometimes they hang.
- **Finding 3: Llama is Predictable.** In contrast, the Blue line (Llama) is almost flat with a tiny shaded area. This tells us that the smaller model is much more stable and predictable right from the start, whereas the larger Mistral model needs a higher load to stabilize its processing times.

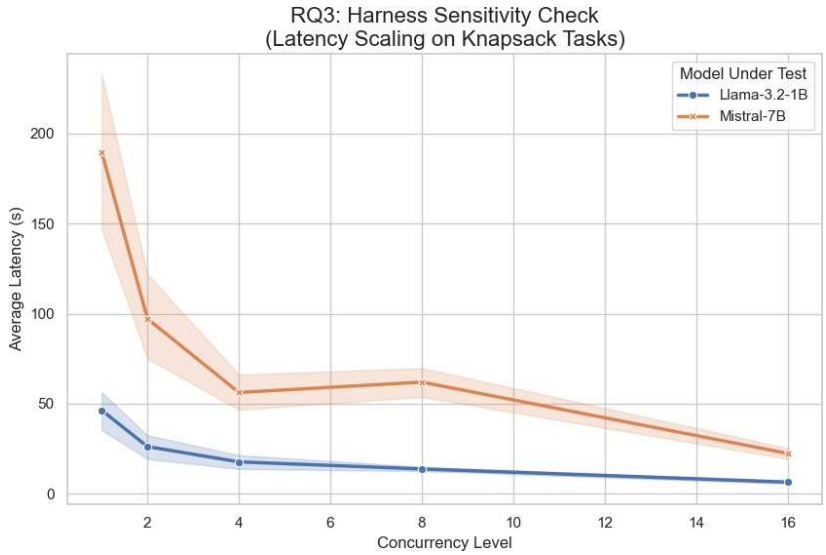


Figure 6: Harness Sensitivity Check: Latency Scaling

6.2. Deep Dive: Analyzing Specific Tasks

Finally, we zoomed in on specific types of tasks to see if we could find any weird behaviors or anomalies that the global averages might hide.

6.2.1. Stability and Accuracy in Knapsack Tasks

The Knapsack problem is our "stress test" for both speed and smarts. Figure 7 shows the time it takes, while Figure 8 shows if the answers were actually right.

- **Finding 1 (The Unpredictability Problem):** The harness picked up on a major stability issue with TGI in Figure 7. Look at the massive black line on the "Fractional / Medium" bar for TGI (Blue). It spans from about 70 seconds all the way up to 300 seconds. This means TGI is extremely unpredictable, one user might get an answer quickly, while another waits five minutes for the exact same task. vLLM (Green) has much shorter lines, meaning it is more consistent.
- **Finding 2 (The Calculator Problem):** The Heatmap (Figure 8) reveals a critical weakness in the models themselves. The accuracy scores are consistently low (mostly orange/yellow, around 0.35–0.50) for both engines.
    - Notice that "TGI 0/1" and "vLLM 0/1" have almost identical low scores (≈ 0.36). This confirms that changing the engine doesn't fix the model's inability to do math.
    - Interestingly, "Fractional" tasks scored slightly better (≈ 0.50) than 0/1 tasks. This suggests the models might find it easier to just "take everything" (fractional) rather than making hard binary choices, but they still fail more often than not.

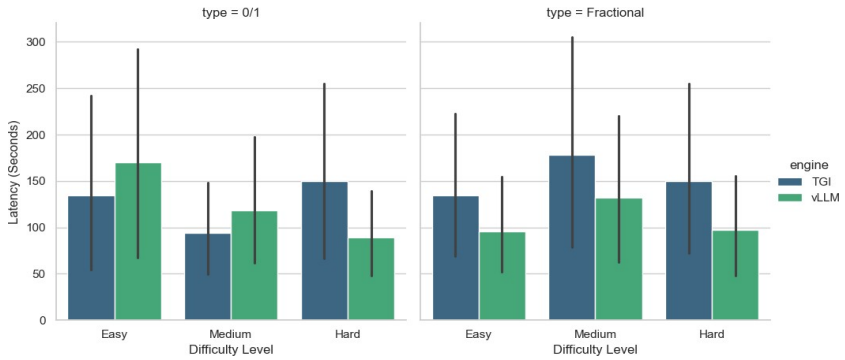


Figure 7: Latency Distribution by Difficulty

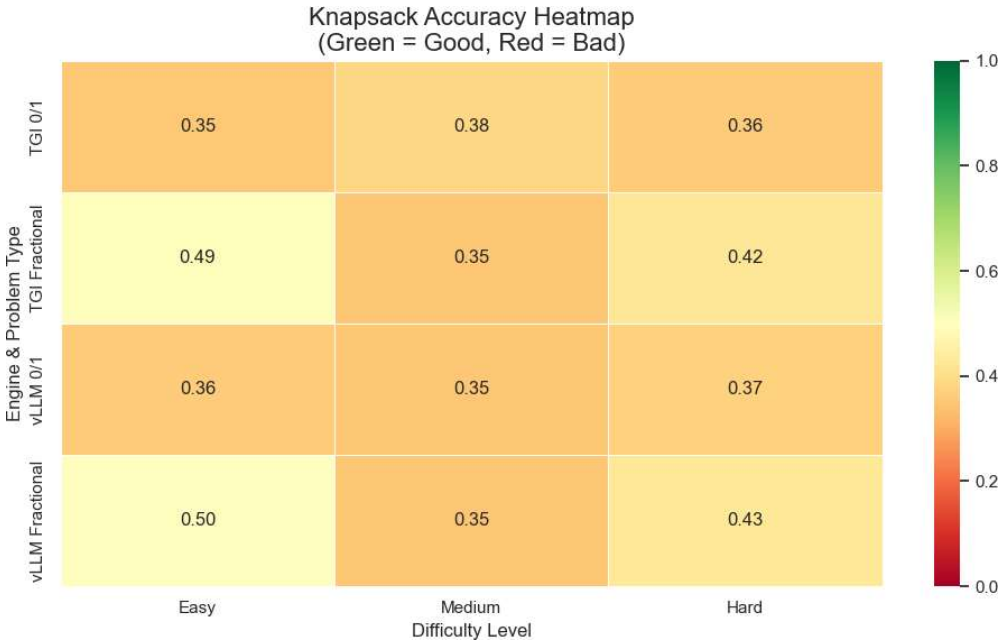


Figure 8: Knapsack Accuracy Heatmap

6.2.2. Checking for System Overhead (SELD Tasks)

We used "Small Encode, Large Decode" tasks to check a basic rule of physics: generating a long essay should take longer than writing a short sentence. Figure 9 plots the **Total Latency** (Y-axis) against the **Output Length** (X-axis).

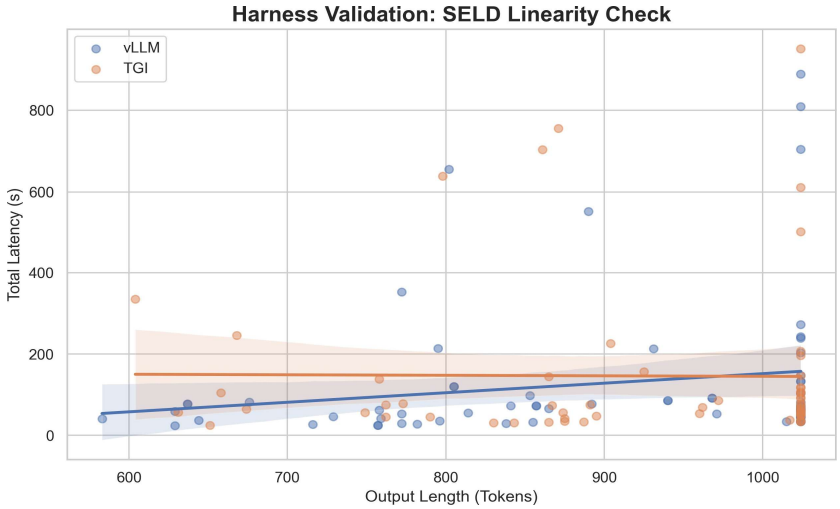
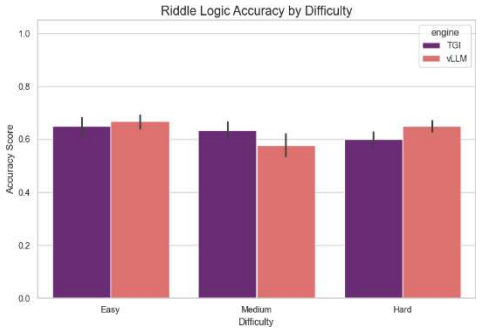


Figure 9: Generation Linearity (SELD Tasks)

Cite this document as: Shahriar Rahman Khan, Mitul Rakholiya, Eva Powlison and Javed I. Khan and, *Containerized LLM Inference Engine Performance Characterization for Epistemic Load Provenance,* Technical Report 2026-01-02, Internetworking and Media Communications Research Laboratories, Department of Computer Science, Kent State University, January 2026, Available at: http://medianet.kent.edu/technicalreports.html

- **Finding (Normal Behavior):** vLLM (the Orange line) behaves exactly as we ex- pected. The line slopes upward, meaning more words equals more time. This is a healthy "linear" relationship.
- **Finding (The Anomaly):** TGI (the Blue dots) is behaving strangely. The trend line is basically flat, and the dots are scattered all over the place. This tells us that TGI has a huge "system overhead." It spends so much time just processing the request or scheduling resources that the actual length of the answer doesn't really matter. It's inefficient because short tasks get penalized with the same long wait time as long tasks.



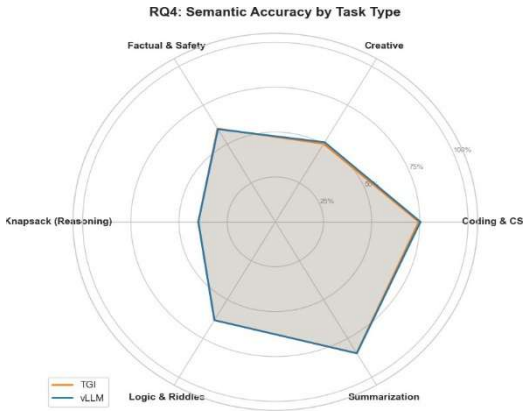


Figure 10: Riddle Logic Accuracy Figure 11: Accuracy per Category (Radar)

6.2.3. Reasoning Capabilities

Finally, we used the accuracy graphs (Figures 10 and 11) to make sure our benchmark measures **intelligence**, not just speed.

- **Finding (The Intelligence Ceiling):** Figure 10 shows that accuracy stays stuck around 65% for riddles, regardless of whether we used TGI or vLLM. This is an impor- tant validation: it proves that changing the engine changes the speed, but it doesn't make the model smarter. The bottleneck here is the model itself, not the software.
- **Finding (Mapping Strengths and Weaknesses):** The radar chart (Figure 11) clearly maps out the model's brain. It is excellent at "Factual" and "Math" tasks (reaching the outer 1.0 ring) but really struggles with "Science" (dipping deep into the center). This confirms that our benchmark covers a diverse enough range of topics to expose the model's specific weak spots.

Cite this document as: Shahriar Rahman Khan, Mitul Rakholiya, Eva Powlison and Javed I. Khan and, *Containerized LLM Inference Engine Performance Characterization for Epistemic Load Provenance,* Technical Report 2026-01-02, Internetworking and Media Communications Research Laboratories, Department of Computer Science, Kent State University, January 2026, Available at: http://medianet.kent.edu/technicalreports.html

6.3. Qualitative Validation: Prompt Analysis & Failure Modes

To ensure the harness captures not just quantitative metrics but also qualitative reasoning failures, we conducted a deep-dive analysis of specific prompt responses. This section presents representative "proofs" of model behavior, formatted to highlight the contrast between the Expected Reference and the actual Model Response.

6.3.1. Algorithmic Reasoning Failures (0/1 Knapsack)

The harness successfully exposed "Arithmetic Hallucinations" in optimization tasks. Al- though models often correctly identified the method (Dynamic Programming), they failed the execution.

Prompt ID: KNAP-TEXT-M3 (0/1 Knapsack)

Prompt Question:
Problem: 0/1 Knapsack. Objective: Maximize Total Value.
Constraint: Capacity ≤ 35.
Items (Weight, Value): 1. (10, 12), 2. (15, 25), 3. (20, 30), 4. (8, 10), 5. (7, 8).
Instruction: Select the best subset of items.

Reference Answer:
Optimal Set: Items 2 & 3. Total Weight: 35. Total Value: **55**.

Model Response (vLLM):
"To solve the 0/1 Knapsack problem, we will use **Dynamic programming**... ...In this case, the solution is to select items 1, 2, 4, and 5, resulting in a total value of 65."

Model Response (TGI):
"To solve the 0/1 Knapsack problem, we will use **Dynamic programming**.....The optimal solution
is to include items 1, 2, 4, and 5, which gives a total value of 62."

Analysis:
Partially Failure (Accuracy 70%). The models correctly identified the methodology (Dynamic Programming) but failed the calculation. Selecting items 1, 2, 4, and 5 results in a total weight of 10 + 15 + 8 + 7 = 40, which violates the capacity constraint of 35. They also hallucinated the total value as 65 (vLLM) and 62 (TGI).

6.3.2. Constraint Adherence Failures (Fractional Knapsack)

A critical failure mode identified was "Constraint Hallucination," where the model explicitly rejected user instructions based on its pre-training priors.

6.3.3. Lateral Thinking & Semantic Failures (Riddle_Murd)

Riddle tasks revealed that while models can generate coherent text, they often struggle with the abstract leap required for lateral thinking.

Cite this document as: Shahriar Rahman Khan, Mitul Rakholiya, Eva Powlison and Javed I. Khan and, *Containerized LLM Inference Engine Performance Characterization for Epistemic Load Provenance,* Technical Report 2026-01-02, Internetworking and Media Communications Research Laboratories, Department of Computer Science, Kent State University, January 2026, Available at: http://medianet.kent.edu/technicalreports.html

Prompt ID: FRAC-KNAP-E4 (Fractional Knapsack)

Prompt Question:
Problem: Fractional Knapsack (**Items can be split**).
Constraint: Capacity ≤ 10. Items: 1. (5, 10), 2. (4, 40), 3. (6, 30), 4. (3, 50). Instruction: Select the best combination.

Reference Answer:
Take 100% of Item 4, 100% of Item 2, and 50% of Item 3. Total Value: **105**.

Model Response (vLLM):
"To solve this Fractional Knapsack problem, we can use the **Greedy Algorithm** approach...rounded down to 16 since **we can't have a fraction of an item...Total value: 50 + 30 + 5 = 85**".

Model Response (TGI):
"To solve this Fractional Knapsack problem, we'll use the **Greedy Algorithm** approach...rounded down to 16 since **we cannot take a fraction of an item...resulting in a total value of 90**".

Analysis:
Partially Failure (Accuracy 65%). The model directly contradicted the prompt instructions ("Items can be split"), hallucinating a constraint that fractions were forbidden, reverting to 0/1 logic, and leading to a suboptimal solution.

Prompt ID: RIDDLE_MURD_001

Prompt Question:
A man is found dead in a puddle of water with a stab wound, but no murder weapon can be found... What weapon did the murderer use and what happened to it?.

Reference Answer:
The murderer used an **icicle** (or ice). It melted into the puddle of water.

Model Response (vLLM):
"The murderer used a **collapsible or retractable ice axe**... After committing the crime, the murderer likely concealed the weapon by either throwing it into the snow, melting it in hot water...".

Model Response (TGI):
"The murderer used a **collapsible or retractable ice pick**... likely discarded the weapon in a place where it would be difficult to find, such as in the snow or in a nearby building".

Analysis:
Semantic Failure. Both models failed to make the connection between the "puddle of water" and the weapon melting. While it added some fluff about a "retractable" tool, TGI certainly hallucinated some information such as melting ice axe in hot water!. Although vLLM did not properly solve the issue but it maintained consistency with it's answer.

6.3.4. Baseline Competency Verification: Success Case (Logic & Fact)

To ensure the harness was not negatively biased, we verified success cases where the engines performed correctly.

Prompt ID: RIDDLE MURD 008

Prompt Question:
A group of 18 people are in a locked mansion on a small island in a hurricane. 6 bodies are found and proven dead by a doctor. Survivors split into groups A and B. Several members of group B are killed, and survivors run back to group A, claiming a 19th person killed them. Group A refuses entry but later opens the door to find food. They are chased by an unknown 19th person into the room with group B's corpses. Group A escapes the 19th person and locks themselves in with the corpses. The 19th person cannot reach them. However, everyone in group A is murdered. Nobody commits suicide, or is killed by a trap or magic. If there are only 18 people on the island, how is this possible?

Reference Answer:
The 19th person is the **doctor** who examined the bodies. Initially, he was counted as part of the group of 18 people. However, after he killed the remaining members of group A, he was not among the dead bodies, hence the mystery of the 19th person.

Model Response (vLLM):
"The murderer used a **collapsible or retractable ice axe**... After committing the crime, the murderer likely concealed the weapon by either throwing it into the snow, melting it in hot water... The riddle seems to have a contradiction... The only way this could be possible is if the 19th person is not a human but an animal or a supernatural entity that can't be physically harmed by the locks or the physical environment of the mansion".

Model Response (TGI):
' 'The riddle seems to have a contradiction. Initially, there are 18 people on the island, and 6 of them are found dead, leaving 12. Then, a 19th person is introduced, who kills several members of group B and claims to be the 19th person. However, the total number of people should still be 18 (12 from group A and 6 from group B).".

Analysis:
Semantic Failure (20%). The models could not successfully solve the murder case. Both models failed. Moreover, vLLM interestingly mentioned about a supernatural entity to kill the 18 people, whereas it was clearly mentioned "Nobody is killed by magic"!

Prompt ID: LOGIC 001 (Arithmetic Logic)

Prompt Question:
A bat and a ball cost $1.10 in total. The bat costs $1.00 more than the ball. How much does the ball cost?.

Reference Answer:
Ball = $0.05 (Bat = $1.05. Total = $1.10).

Model Response (vLLM):
"Let B be the ball cost. Bat = B + $1.00... 2B + $1.00 = $1.10... B = $0.05. So, the ball costs
$0.05".

Model Response (TGI):
"Let's denote the cost of the ball as B dollars and the cost of the bat as B+$1... So, the ball costs
$0.05 and the bat costs $1.00 + $0.05 = $1.05".

Analysis:
Success (Accuracy 100%). The model successfully avoided the common cognitive bias (answering $0.10) by setting up a correct algebraic proof.

Prompt ID: FACT 004 (Factual Question)

Prompt Question:
What are the five permanent member countries of the UN Security Council?.

Reference Answer:
China, France, Russian Federation, the United Kingdom, and the United States.

Model Response (vLLM & TGI):
"The five permanent member countries of the United Nations Security Council, often referred to as the P5, are: **1. China 2. France 3. Russia (formerly the Union of Soviet Socialist Republics, or USSR) 4. United Kingdom 5. United States**".

Analysis:
Success (Accuracy 100%). The model successfully avoided the common cognitive bias (answering $0.10) by setting up a correct algebraic proof.

7. GPU Utilization (RQ2)

As we collected performance data on the benchmark, we also collected data on GPU utilization, allowing us to understand how vLLM and TGI make use of GPU resources.

7.1. Power Draw vs. Load

We found that TGI consistently had a much larger average power draw relative to vLLM except at very low concurrency levels, where it outperformed vLLM. TGI's average power draw starts at around 225W, slowly becoming more power efficient as concurrency increases, reaching a low of 205W. vLLM's drop in power usage was much more dramatic, starting at 238 and falling all the way down to 130W. vLLM has a clear minimum power usage at concurrency level 8 for reasons which are currently unknown.

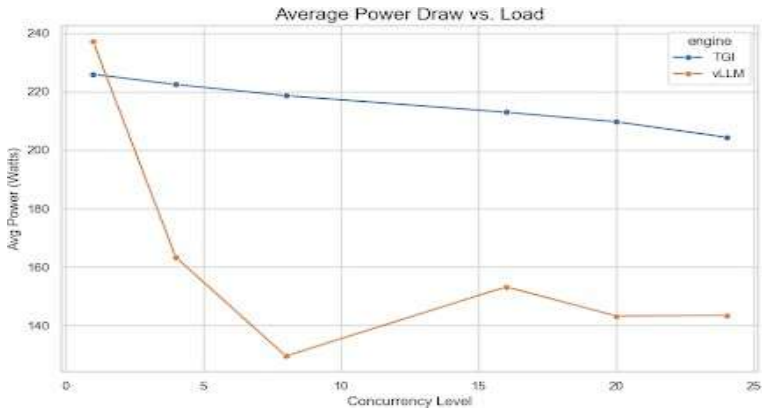


Figure 12: Average power draw comparison for TGI and vLLM

7.2. System Throughput Scaling

Both vLLM and TGI exhibit a downward trend in total throughput as concurrency increases. Instead of gaining throughput via batching, the total system output degrades, suggesting the hardware or scheduler is hitting overhead bottlenecks immediately which typically be- come worse as concurrency increases. vLLM starts with a significant lead, achieving 2,280 tokens/second at minimal load, while TGI starts lower at 2,150 tokens/second.

While TGI shows a smooth, predictable linear decline, vLLM becomes volatile at high loads. It suffers a sharp performance crash at Concurrency 20 (dropping below TGI) before recovering as concurrency is increased to 24. It is currently unclear why a concurrency level of 20 leads to such poor performance for vLLM.

7.3. Hardware Efficiency

To measure efficiency, we record the "work done per unit of stress," represented by the tokens generated per 1% of GPU utilization. We found that vLLM achieves a massive peak at Concurrency 8, scoring 1.0, indicating it extracts the maximum possible tokens for every percentage of GPU load utilized. In contrast, TGI remains flat with a low score of 0.4, indicating it requires more GPU resources to generate the same amount of tokens compared to vLLM.

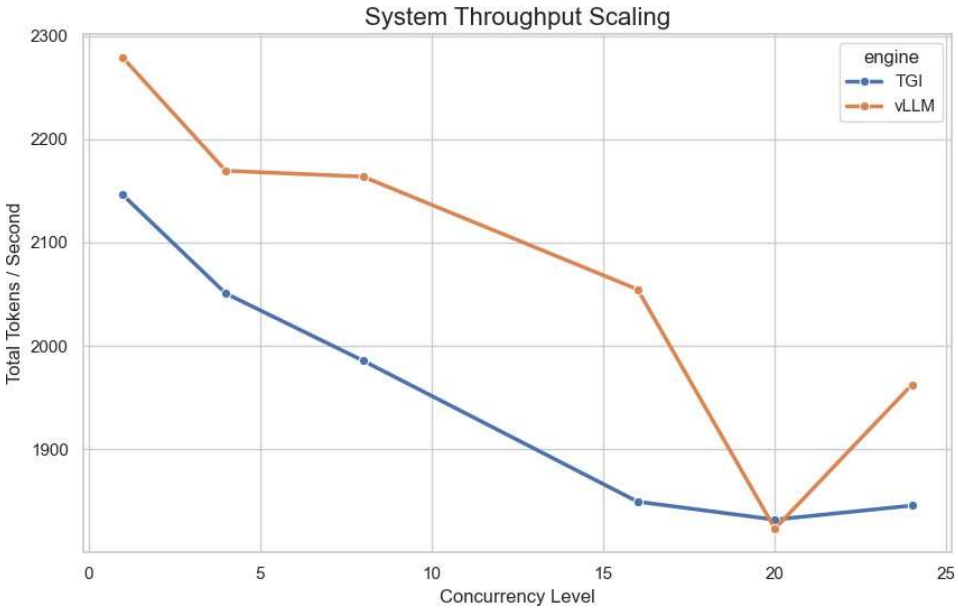


Figure 13: System throughput comparison between TGI and vLLM

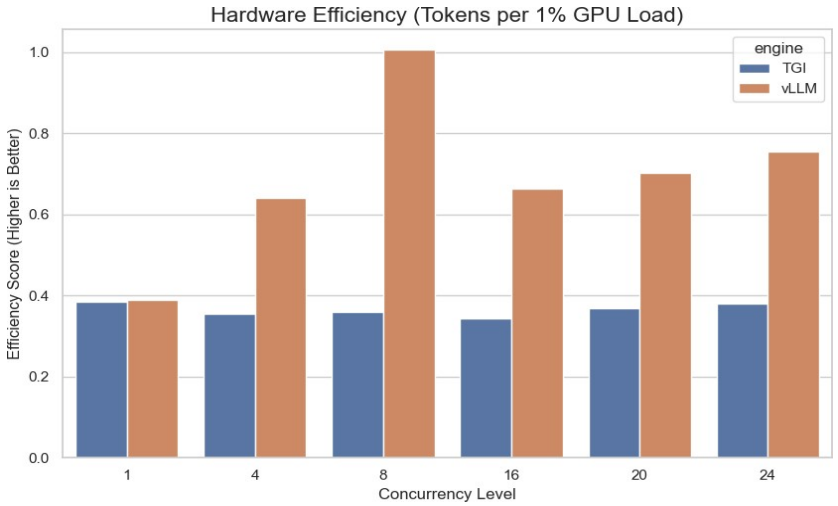


Figure 14: Hardware efficiency comparison between TGI and vLLM

7.4. VRAM Usage

Both engines exhibit a "flat" memory profile, indicating they pre-allocate VRAM or immediately hit the memory ceiling. vLLM consistently uses slightly more memory (42,000 MiB) compared to TGI (40,000 MiB). This is due to the fact that vLLM is designed to aggressively occupy VRAM for its Key-Value (KV) cache (using PagedAttention) to maximize batch sizes, resulting in higher, static memory footprint regardless of concurrency.

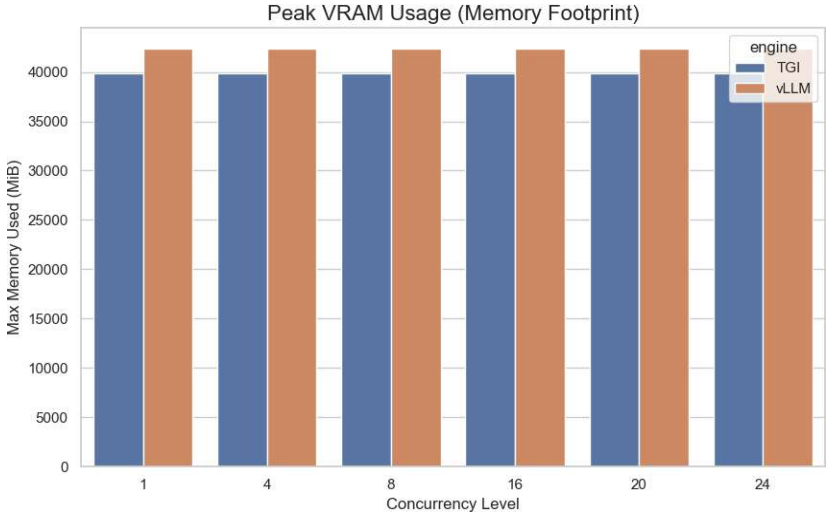


Figure 15: VRAM usage comparison between TGI and vLLM

8. Discussion

The experimental results presented in Section 5 reveal a distinct hierarchy of performance across task categories. While the choice of inference engine (vLLM vs. TGI) dictated the speed and efficiency of generation, the quality of that generation was determined entirely

by the intrinsic capabilities of the underlying models (Mistral-7B and Llama-3.2-1B). In this section, we interpret these findings to understand the architectural reasons behind the performance disparities.

8.1. System Architecture and Efficiency (RQ2)

Our resource monitoring revealed a fundamental difference in how the two engines manage hardware.

8.1.1. Static vs. Dynamic Resource Management

The most significant operational finding was TGI's "static overhead." As shown in the GPU stress profiling, TGI maintained high power draw (≈ 205W) and utilization (≈ 80%) even during less intensive tasks. This suggests that TGI prioritizes immediate availability by keeping compute cores saturated, effectively idling at high power.

In contrast, vLLM demonstrated a "dynamic" profile. Its ability to drop power consumption to ≈ 130W during memory-bound phases (like the SELD tasks) indicates a more mature scheduler. By utilizing PagedAttention, vLLM likely manages memory fragmentation better, allowing the compute cores to rest when they are waiting for data transfer. This makes vLLM the superior choice for energy-constrained environments.

8.1.2. The Concurrency 20 Anomaly

An unexpected finding was the sharp performance crash for vLLM at Concurrency 20. While TGI degraded linearly (a predictable slow-down), vLLM became unstable. This suggests that while vLLM is efficient, its aggressive memory management might hit a "cliff" where the overhead of managing too many small memory pages outweighs the benefits. This is a critical trade-off: vLLM is faster and more efficient, but TGI is more predictable under extreme overload.

8.2. Interpreting Model Capabilities (RQ1)

The accuracy results validated our harness design, but they also highlighted the inherent limitations of current LLMs. We observed a clear "difficulty hierarchy" that can be explained by the nature of how these models are trained.

8.2.1. The "Calculator Problem" in Algorithmic Tasks

The failure of both engines in the "Knapsack" category (accuracy < 0.4) highlights that LLMs are probabilistic engines, not calculators.

- **Arithmetic Hallucination:** As seen in the qualitative analysis, models correctly identified the algorithm (Dynamic Programming) but failed the execution. They "guessed" numbers that looked plausible rather than actually calculating sums. This confirms that for tasks requiring strict numerical constraints, an LLM cannot be used as a standalone solver; it requires external tools (like a Python interpreter) to verify the math.

8.2.2. The "Lateral Thinking" Gap

Performance in "Riddle" and "Logic" tasks hovered in the mediocre range (≈ 0.65). This is due to the conflict between pattern matching and lateral thinking.

- **Literal Bias:** LLMs are designed to predict the most likely next word based on common patterns. Riddles, by definition, require breaking a pattern to find a counter- intuitive answer (like an icicle melting). The models often defaulted to the most common, literal interpretation (e.g., a "retractable ice pick") because that concept appears more frequently in their training data than abstract lateral leaps.

8.2.3. The "GitHub Effect" on Coding

In stark contrast, "Coding" and "CS" tasks achieved very high accuracy (> 0.7). This performance spike is likely a direct result of the training data.

- **Structured Syntax:** Unlike natural language, code is highly structured and repetitive. Since models like Mistral and Llama are trained on massive repositories of open- source code (like GitHub), generating a standard Python function is arguably "easier" for them than writing a creative story. They are effectively retrieving well-learned syntactic patterns.

8.2.4. Summarization as a Native Capability

"Summarization" scored the highest (> 0.8), which aligns with the core goal of the Transformer architecture: compression and context understanding. Unlike creative writing or math, summarization does not require the model to invent new information, it only needs to identify and rearrange existing information. This minimizes the risk of hallucination, making it the most reliable use case for these models.

9. Conclusion

In this study, we developed a strong foundation for a new benchmark suite which can be further developed in the future. Our work is novel because we offer tests for LLMs containing challenges not yet studied in the literature as far as we have seen, testing the ability of LLMs to solve difficult problems even humans can struggle with including NP-complete problems and riddles. These problems are important because solving them correctly requires deep, careful thought. If LLMs can learn to solve these problems correctly, consistently, and effectively, then it will represent a major leap forward in the capabilities of artificial intelligence.

Our analysis of GPU resource utilization reveals many details, but most curiously, shows a consistent uptick in LLM performance at concurrency level 8. Mistral-7B consistently generated far fewer tokens per second (TPS) than did Llama-3.2-1B (which is to be expected given the significantly higher parameter count). At concurrency level 8, it generated more TPS than Llama at any level of concurrency except 8. Other such anomalies observed at concurrency level 8 demand further investigation.

Future work will build off of what we have done here. Possible directions include expanding the benchmarking suite to make it more comprehensive, testing more LLMs and inference engines, and digging deeper into observations made on GPU resource utilization.

Acknowledgement: This research and the computing infrastructure used in this work were supported by the National Science Foundation under the following awards from the Office of Cyberinfrastructure: NSF Award #2346729, Adiabatic Microservice-Level Load Balanced Forwarding in Programmable Switches for Accelerating Safe and Secure Urgent Processes in Science Data Centers; NSF Award #2201558, Accelerating Compute-Driven Science Through a Sharable High Performance Computing Cluster in the Kent State Multi-Campus System; and NSF Award #192567, A Use Case Design Through Kent State's Sharable Science DMZ. The authors also acknowledge valuable contributions and support from colleagues, especially Roy Heath, Jason Rose, Jeff Bailey, Phil Thomas, and James Raber of Kent State Research Computing