**Green Inference**

**A Systematic Framework for Measuring the Energy Efficiency of Local LLM Serving**

## **Project Relevance to Formal Evaluation Metrics**

## **1. Problem, Research Question, and Relevance to AI/DevOps**

The project addresses a distinct problem: the lack of public, reproducible benchmarks for the energy consumption of AI models on widely available consumer hardware. This absence of data prevents students, developers, and researchers from making evidence-based decisions regarding the efficiency and sustainability of their tools.

The central research question is therefore: How do different serving configurations, specifically batch size and caching, affect the performance and energy efficiency of Small Language Models (SLMs) on consumer-grade GPUs?

This investigation is directly relevant to the field of AI/DevOps. The project implements core principles of this discipline by building an automated pipeline for testing and monitoring AI models (MLOps). The focus on measuring resource usage and optimizing for lower energy costs directly relates to the operational goals of efficiency and financial management (FinOps) in a production AI system.

## **2. Data Collection, Source, and Preprocessing**

The project generates its data through a series of controlled experiments. The raw data collected is quantitative and includes four primary types:

1. GPU Power Draw (Watts)
2. GPU VRAM Usage (MB)
3. Request Latency (seconds)
4. Generated Token Count

The quality and integrity of this data are ensured through two key measures. First, the workload is rigorously standardized. We extracted a diverse set of **300 prompts** from the well-regarded **databricks-dolly-15k** dataset, which contains high-quality, human-written instructions. These prompts were categorized by complexity into three distinct sets: **100 small, 100 medium, and 100 large queries**. Each model was then benchmarked against these standardized sets. Second, the entire experiment is run within an isolated **Docker container** to eliminate environmental variables and ensure that all measurements are consistent and repeatable.

A critical step in the methodology is the preprocessing of this raw data. The script aggregates high-frequency power readings to calculate an average power draw for each test run. This average is then used with the total test duration to compute the total energy consumed in **Joules**. Finally, this energy value is normalized by the number of tokens generated to produce our core efficiency metric: **Energy per Token (Joules/token)**. This process transforms raw system telemetry into a meaningful and comparable measure of performance.

## **3. Methodology and Analytical Approach**

The core of this project is a **controlled experiment**, a methodology chosen for its ability to reliably isolate and measure the effect of specific changes on system performance. Our experimental design is structured as follows:

* **Independent Variables:** These are the factors we intentionally manipulate to observe their effect. They include the GPU hardware (RTX 1650 vs. RTX 3050 Ti), the model size (0.5b, 1.5b, 7b), and the optimization settings (batch size, caching, and parameter tuning).
* **Dependent Variables:** These are the key metrics we measure to quantify the outcome of our changes. The primary dependent variables are performance (measured in **Tokens per Second**) and efficiency (measured in **Energy per Token**), supplemented by latency and VRAM usage.
* **Controls:** To ensure fair and repeatable comparisons, all experiments are conducted within an isolated **Docker container**. This provides a consistent software and driver environment, eliminating system-level variables that could otherwise influence the results.

Our analytical approach is one of **comparative benchmarking**. By systematically testing every combination of our independent variables, we produce a structured dataset that allows for a direct, empirical comparison of each configuration's trade-offs. This method is essential for our research question, as it provides the concrete evidence needed to draw justifiable conclusions about which setups offer the optimal balance of speed, responsiveness, and efficiency for a given task.

**4. Workflow and Reproducibility**

The experimental workflow was designed to be both logical and fully reproducible, ensuring the integrity of the results. The entire process is encapsulated within a framework that moves from environment setup to final analysis.

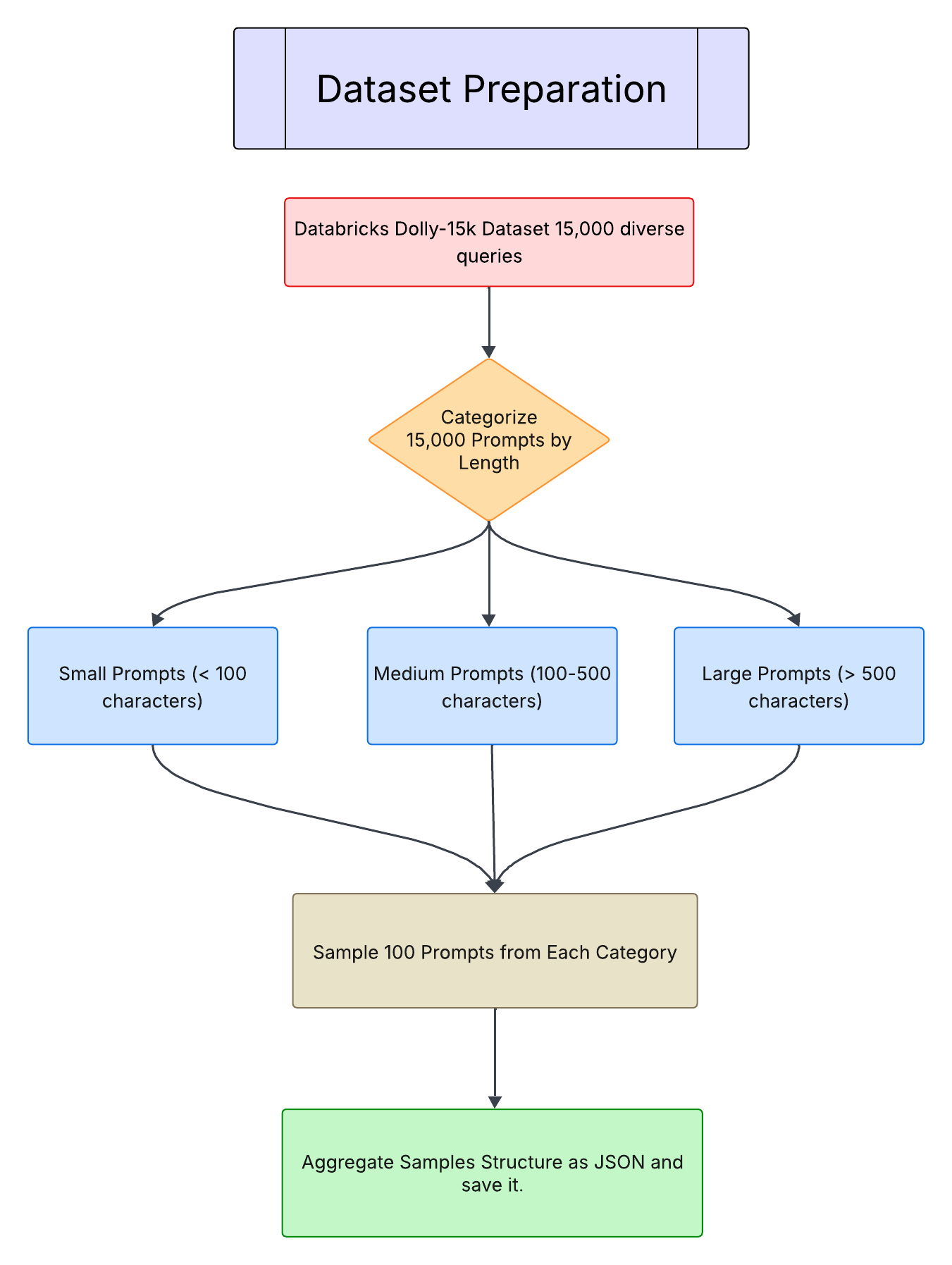
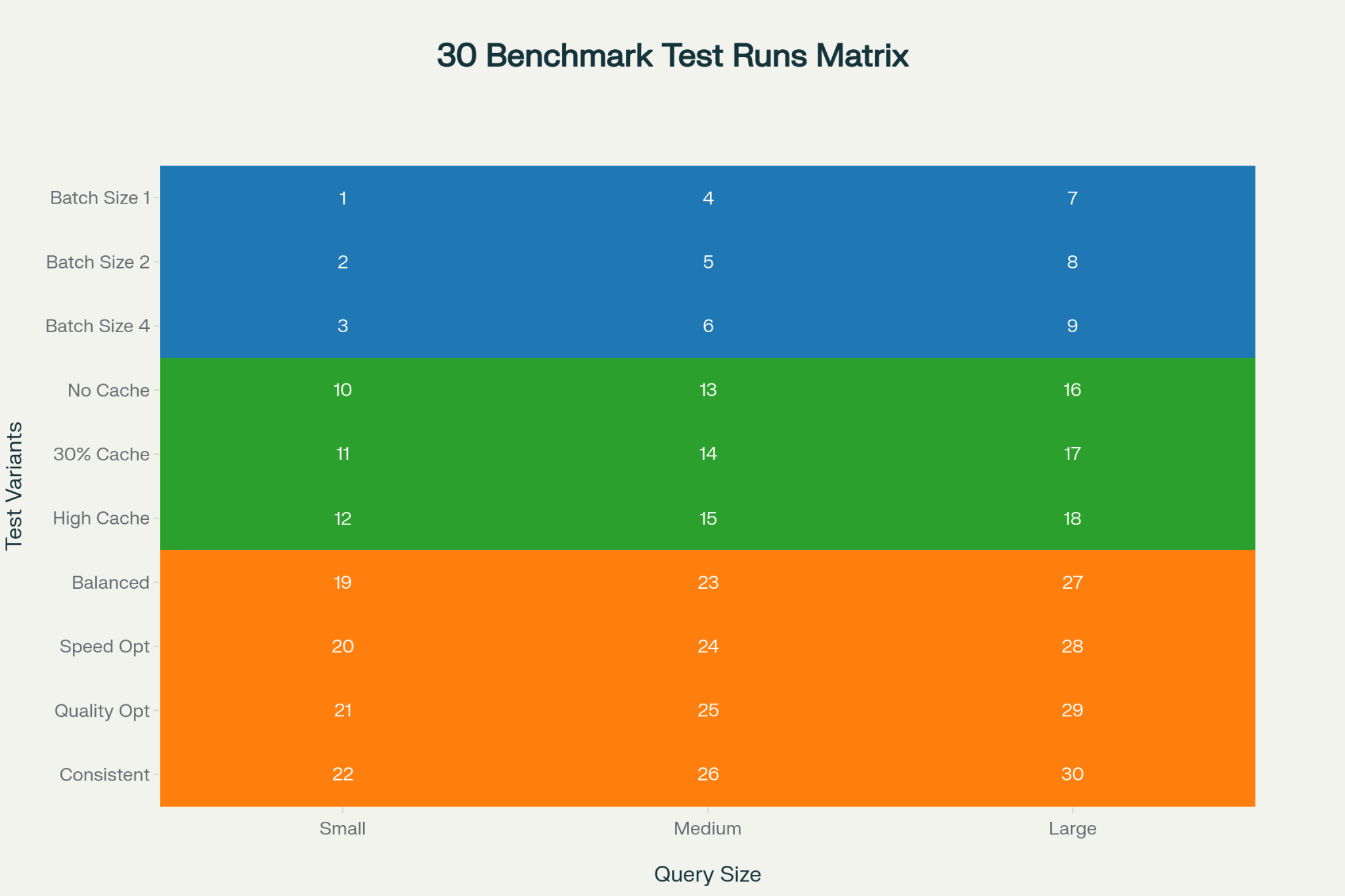
The workflow begins with the **environment setup**. A **Docker container** is used to create an isolated and consistent environment. This container includes the specific Python version, deep learning frameworks (like PyTorch), and the exact NVIDIA drivers required to run the models. This critical first step eliminates variations from the host system's configuration, ensuring that any performance differences observed are due to the variables being tested, not the environment.

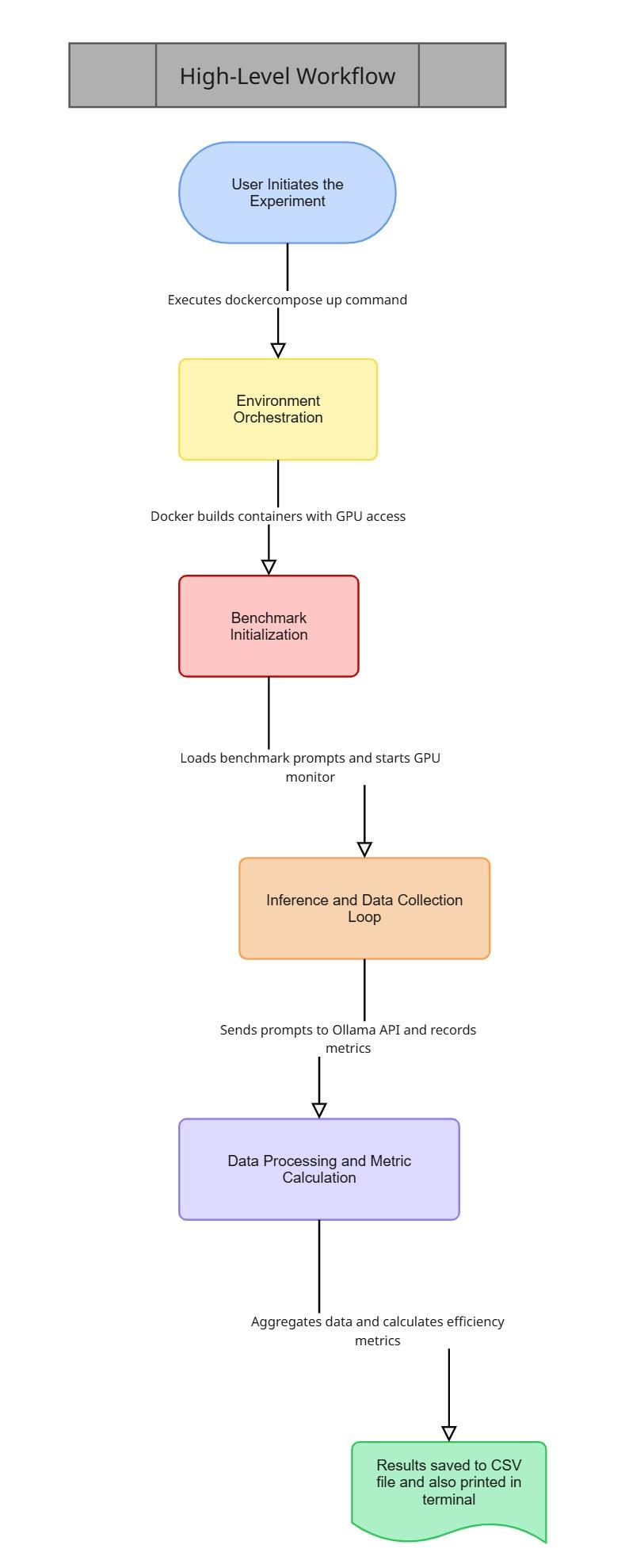
The second stage is **workload execution**. A master Python script orchestrates the entire benchmarking process. This script systematically iterates through every defined configuration:

1. It loads a specific **Qwen2 model** (0.5b, 1.5b, or 7b).
2. It cycles through each **test methodology** (Batching, Caching, Parameters).
3. For each methodology, it iterates through the corresponding **variants** (e.g., Batch Size 1, 2, 4).
4. During each run, the script feeds the standardized prompts from the **databricks-dolly-15k** dataset to the model.

Simultaneously, a background process queries the NVIDIA Management Library (**nvidia-smi**) at a high frequency to log raw telemetry data, including **GPU power draw** and **VRAM usage**, into a temporary file. The script also records the **latency** for each query and the total **token count** generated.

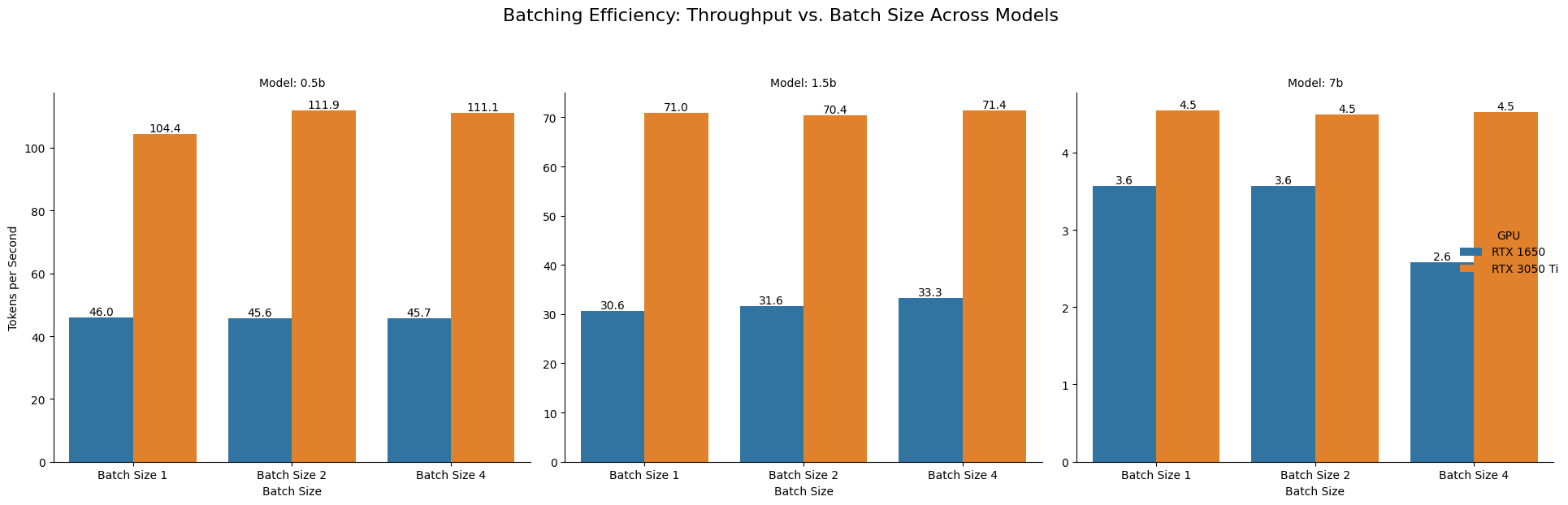
The final stage is **data processing and aggregation**. Once a test run is complete, another script processes the raw telemetry and performance logs. It calculates the average power draw, total energy consumption (in Joules), and key performance metrics. These processed results are then compiled into a structured **CSV file** for each major configuration (e.g., **focused\_benchmark\_qwen2\_0.5b\_RTX\_1650.csv**), which serves as the clean, final dataset for analysis. This structured approach ensures that any researcher with access to the Dockerfile and the scripts can replicate the experiment and verify the findings.

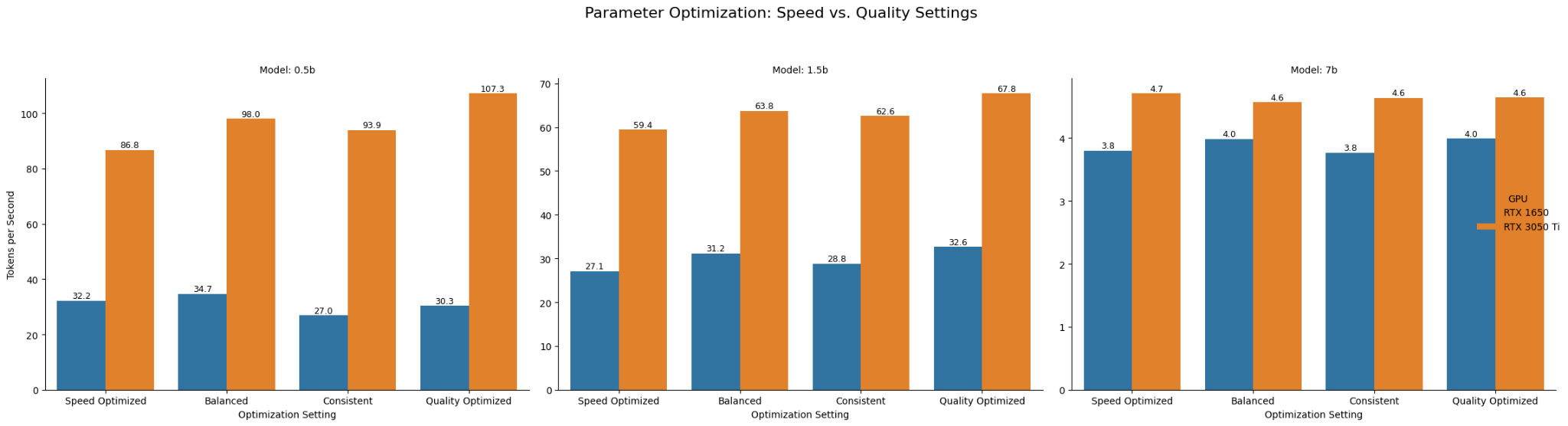
****

****

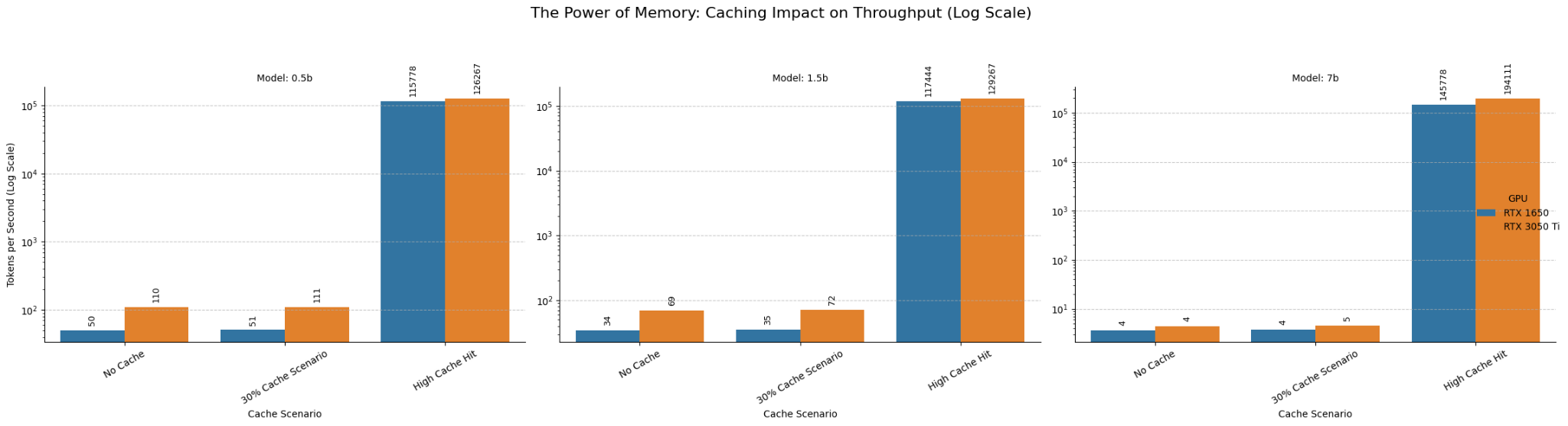
### **5. Presentation of results with visualizations**

The results of the analysis are presented through a suite of carefully selected visualizations, each designed to answer a specific question about the system's performance and efficiency. The use of data visualization is critical for transforming complex numerical data into clear, interpretable insights.

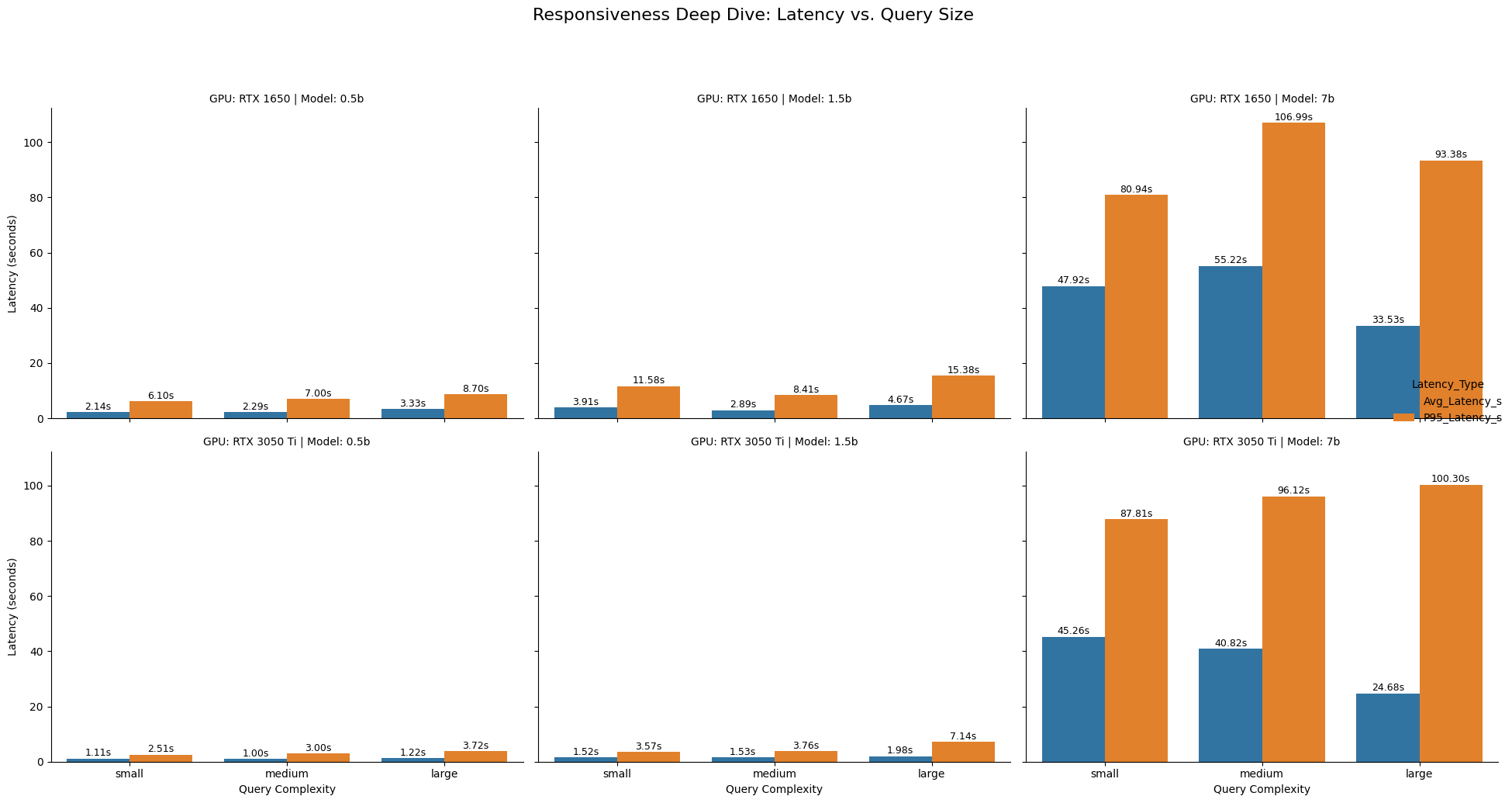
The primary tool used for generating these visualizations is the **Seaborn** library in Python, valued for its ability to create statistically informative and aesthetically pleasing charts. The results are presented as follows:

This bar chart compares tokens per second at batch sizes 1, 2, and 4 for both GPUs across model sizes, establishing how batching scales under different compute/memory headrooms. The 0.5B model shows a clear throughput rise from batch 1→2 on the RTX 3050 Ti with a plateau by batch 4, while the RTX 1650 shows minimal gain, indicating early saturation; the 1.5B model exhibits diminishing returns, and the 7B model shows no benefit or a decline on the RTX 1650 at higher batch, revealing hardware saturation effects. Throughput (tokens/sec) is chosen because it directly measures serving capacity and parallelism effectiveness across hardware and workloads.

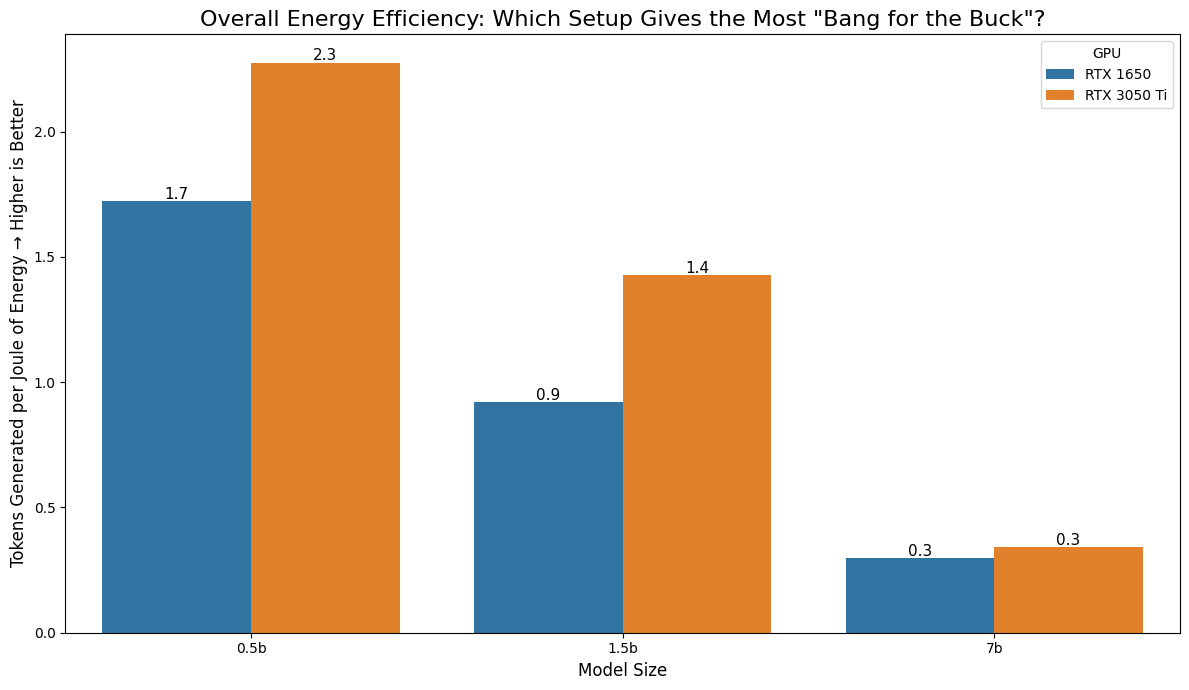
This grouped bar chart contrasts decoding profiles across models and GPUs to isolate software‑side tuning effects on tokens per second, showing that small/medium models respond to settings while large models remain hardware‑bound. The RTX 3050 Ti consistently leads on 0.5B/1.5B (2–3× faster than RTX 1650), whereas the 7B model’s throughput clusters tightly regardless of profile, indicating that parameter tweaks cannot overcome compute/memory bottlenecks at this scale. This figure complements Figure 1 by separating batching (parallelism) from decoding (generation behavior) to inform practical tuning levers.



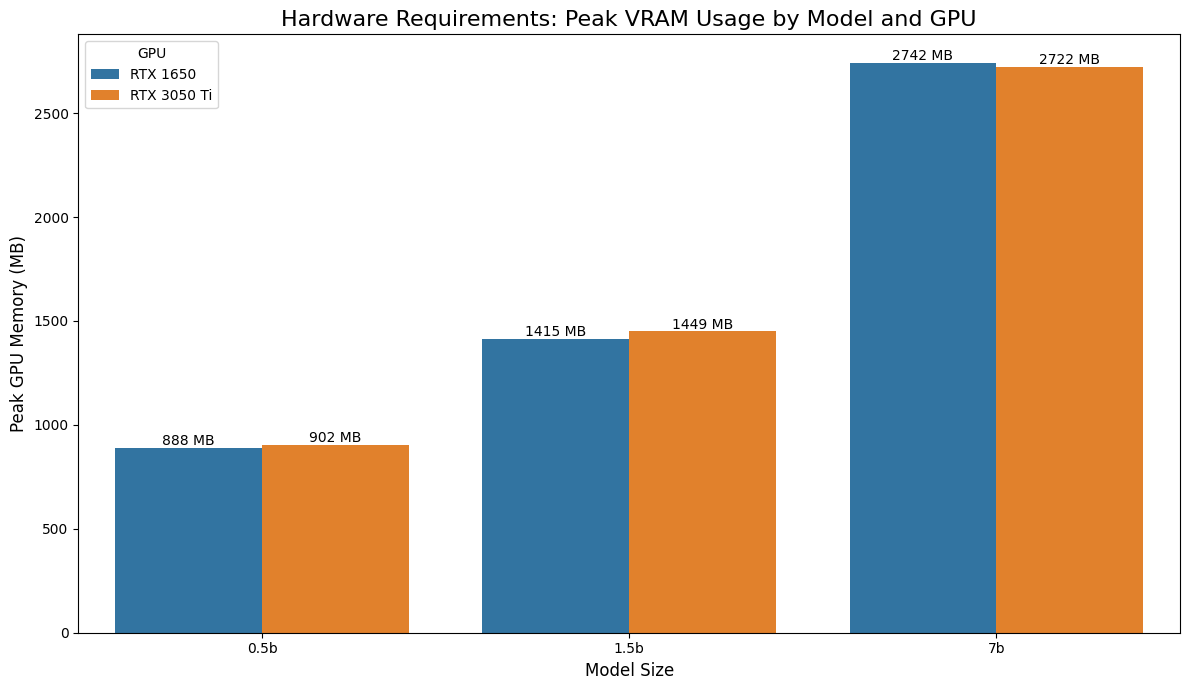
This three‑panel chart reports tokens per second under No Cache, ~30% Cache, and High Cache hit scenarios for each model size, using a log y‑axis to reveal multiplicative gains of two to three orders of magnitude at high hit rates. The 0.5B and 1.5B models achieve ~150–200× gains under high cache, and the 7B model reaches ~2,000–3,000×, while ~30% cache provides only marginal improvement, demonstrating a threshold effect where caching becomes transformative only at high hit rates. The log scale is appropriate given the extreme range, and throughput here reflects real‑world effects of KV/prompt caching policies on serving capacity.



This panel illustrates response time distributions, emphasizing both mean and tail latency to reflect user‑experienced responsiveness under different conditions. The 0.5B and 1.5B models remain sub‑~20s with tight tails suitable for interactive use, whereas the 7B model exhibits ~40–110s latencies with P95 exceeding ~100s, indicating non‑interactive behavior even on the RTX 3050 Ti in typical scenarios. Reporting P95 complements average latency by capturing tail risk and worst‑case experience crucial for production SLAs.



This chart normalizes performance by power to present efficiency, revealing a “sweet spot” at smaller models on modern GPUs and a steep efficiency decline at 7B. The analysis shows that quality‑optimized decoding surprisingly provides the best energy efficiency across sizes, and that efficiency leadership can switch with workload scale (older GPUs may be competitive on tiny models, while newer architectures dominate at heavier loads). Tokens/Joule and Joules/token are deployment‑relevant metrics because they approximate operating cost and environmental impact at inference time.



This bar chart reports peak VRAM by model size and GPU, showing sub‑linear memory scaling with parameters and minimal differences between RTX 1650 and 3050 Ti, which underscores that model architecture, not GPU generation, primarily determines footprint. The 7B model approaches ~2.7–2.8 GB, implying a practical minimum of ~4 GB VRAM and a recommendation of ≥6 GB for stability and batching headroom, which sets hard deployment gates regardless of throughput considerations. Peak VRAM is included because it defines feasibility boundaries; if memory exceeds device capacity, the configuration is non‑deployable.

## **6. Interpretation of findings and meaningful insights**

* Batching is beneficial only with headroom: Modest batching improves throughput for 0.5B on RTX 3050 Ti and provides limited gains for 1.5B, but offers no benefit—and can harm performance—on 7B or on constrained GPUs like RTX 1650 at higher batch sizes, indicating early saturation and memory bandwidth/allocator pressure. This pattern implies selecting conservative batch sizes for small/medium models on stronger GPUs and avoiding aggressive batching on large models or weaker GPUs. The observed plateau/decline highlights when parallelism ceases to translate into throughput due to compute or memory bottlenecks.
* Decoding parameters matter for small/medium models, not for large: Speed/Balanced/Consistent/Quality profiles meaningfully shift 0.5B/1.5B throughput, particularly on RTX 3050 Ti, but barely move 7B, confirming that software‑side tuning cannot offset hardware limits at large scale. Practically, parameter tuning should be used to dial speed/quality trade‑offs for 0.5B–1.5B, while 7B requires model compression, distillation, or hardware changes rather than decoding tweaks. This separation of concerns helps prioritize engineering effort where it yields measurable returns.
* Caching is the dominant multiplicative lever: High cache hit rates yield ~150–200× gains on 0.5B/1.5B and ~2,000–3,000× on 7B, dwarfing the benefits of batching or decoding settings, whereas ~30% caching delivers little improvement, indicating a threshold beyond which caching becomes transformative. For repeated or templated workloads, cache engineering (prompt/result/KV cache) is non‑negotiable to achieve acceptable capacity and latency, especially for 7B deployments on consumer GPUs. The log‑scale visualization is necessary to faithfully depict these order‑of‑magnitude effects.
* Latency defines UX regimes: 0.5B and 1.5B deliver interactive responsiveness with modest tails, while 7B shifts to batch‑style latency windows with large P95 values, changing application fit from real‑time chat to asynchronous generation. The counter‑intuitive pattern where large queries occasionally outperform medium for 7B suggests allocator/runtime dynamics worth future profiling but does not change the overall non‑interactive regime at this scale. Production SLAs should therefore map model size to interaction mode rather than attempting to force 7B into low‑latency use cases on this hardware tier.
* Energy efficiency has a sweet spot and a cliff: Efficiency peaks for small models on modern GPUs and degrades substantially as parameters increase, with quality‑optimized decoding repeatedly ranking best on Joules/token despite lower raw throughput, implying that “going faster” can carry an energy premium. From 0.5B→7B, energy per token rises multiple‑fold, so deployments must justify 7B with task‑critical quality or rely on caching/quantization to control cost and footprint. These findings support a green‑by‑design strategy that prioritizes smaller models where feasible.
* VRAM is a hard constraint: While throughput and energy can be tuned, peak memory draws a clear feasibility boundary; 7B sits near ~2.7–2.8 GB, making ≥4 GB VRAM a minimum and ≥6 GB advisable for stability and batching on consumer devices. Because VRAM consumption is largely architecture‑driven and similar across the two GPUs, memory planning must precede performance optimization in the deployment checklist. This framing prevents engineering cycles on configurations that cannot run reliably.

### **7. Evaluation Metrics and Their Significance**

To conduct a comprehensive analysis, a set of appropriate evaluation metrics was used, each chosen to measure a specific aspect of system behavior.

* **Tokens per Second (Throughput):** This is the primary metric for **raw performance**. It measures the rate at which the model generates text. Its significance lies in its direct correlation to the system's processing speed; a higher value indicates a more powerful and faster system, capable of completing tasks in less time.
* **Latency (Average and P95):** This metric is crucial for evaluating the **user experience**. It measures the delay between a request and a response. While average latency provides a general sense of responsiveness, the **P95 Latency** is more significant as it captures the "worst-case" experience for 95% of users. A low P95 latency is essential for real-time, interactive applications like chatbots.
* **Energy per Token (Joules/Token):** This is the core metric for **efficiency**. It measures how much energy is consumed to generate a single token. Its significance is immense for real-world deployment, as it directly relates to the operational cost (electricity bills) and thermal output of the system. A lower value indicates a more efficient and sustainable configuration.
* **Peak VRAM Usage (MB):** This metric measures the maximum amount of dedicated GPU memory required to run a model. Its significance is that it defines the **minimum hardware requirement**. A model is simply unusable if its peak VRAM usage exceeds the capacity of the installed GPU, making this a critical go/no-go metric.