MLperf v3 KV cache proposal

Date: November 5, 2025
Subject: A detailed technical explanation of the `kv-cache.py` benchmark for system architects and performance engineers.

Authorship Note: The benchmark architecture, scenario planning, and debugging were led by Hazem Awadallah hazem_awadallah@kingston.com decisions; AI tooling was used selectively to draft code under that direction.

---

1. Introduction: Solving the LLM Memory Problem

At the heart of an LLM's ability to understand context is the attention mechanism, which relies on a data structure called the KV Cache. During inference, LLMs generate text one token at a time in a process called autoregressive decoding. To generate the next token accurately, the model must consider all the preceding tokens in the sequence.

Instead of wastefully re-calculating the contextual meaning of the entire sequence for every new token, the model uses the KV Cache. This cache stores the intermediate attention data, specifically, the "Key" and "Value" vectors, for every token already processed. When generating a new token, the model reuses these cached values, which dramatically reduces computation and speeds up response generation.

The bottleneck emerges from the cache's memory consumption. The size of the KV Cache grows linearly with the length of the token sequence. For applications involving long context windows, such as multi-turn conversations or analyzing large documents, the cache can become enormous, quickly consuming the limited and expensive high-speed memory (VRAM) on a GPU

This creates a critical system design challenge: **where do you store the KV cache?** Offloading it from expensive GPU VRAM to more abundant CPU RAM or even NVMe storage is a cost-effective solution, but it introduces latency. Moving data is always slower than accessing it locally.

This benchmark was designed to solve this exact problem. It provides a sophisticated, configurable tool that allows system architects to **quantify the performance trade-offs of different storage tiers.** By simulating a realistic multi-tenant inference workload, it helps you answer critical questions:

- How much GPU VRAM and CPU RAM do I need for my target user load?
- Is my NVMe drive fast enough to handle the spillover?
- What is the real-world latency impact of offloading to a specific tier?
- Where is the bottleneck in my system: the GPU, the CPU, or the storage?

How to Use This Benchmark Properly:
This is not a simple "pass/fail" test. It's a diagnostic tool.

1. **Start with the `storage-only` workload.** This isolates your storage device and tells you its absolute performance limits. If your drive fails this test, it will be a bottleneck in any multi-tier configuration.
2. **Run the `cpu-storage` and `gpu-cpu-storage` tests.** These represent realistic production scenarios. Compare the latency and throughput to understand the value of each tier.
3. **Use the `autoscale` workload.** This is the most valuable test. It automatically finds the maximum number of concurrent users your specific hardware configuration can support before performance degrades unacceptably. Use this number to configure your production environment.

---

2. Recommended Benchmark Invocations

Here are the specific commands to run for a thorough analysis of your system. These examples assume you are testing the `llama3.1-8b` model and have a cache directory at `/mnt/nvme`.

Step 1: Isolate and Test Storage Performance

This command uses a minimal CPU RAM budget (0.5 GB) to force all I/O to your NVMe drive. It establishes the performance baseline for your storage. Using a fixed `--seed` ensures that the "random" workload is identical every time, making results comparable.

```
# Test 1: Storage-Only Workload
python3 kv-cache.py \
    --model llama3.1-8b \
    --num-users 50 \
    --duration 180 \
    --gpu-mem-gb 0 \
    --cpu-mem-gb 0.5 \
    --generation-mode realistic \
    --cache-dir /mnt/nvme \
    --seed 42 \
    --output results_storage_only.json
```

What to look for: Check the **NVMe Throughput** in the `STORAGE PERFORMANCE ASSESSMENT` section of the output. For this saturation test, high latency is expected and acceptable; the key metric is the sustained **tokens/sec** your drive can handle. This value represents your storage's performance ceiling. Compare it across different drives to find the best one for your workload.

Step 2: Test a Realistic Multi-Tier Configuration

This command simulates a production environment with a full three-tier hierarchy. It uses a larger, more realistic CPU memory budget and enables the GPU if available.

```
# Test 2: Full Three-Tier Realistic Workload
# (Set --gpu-mem-gb to your available VRAM, or 0 if none)
python3 kv-cache.py \
    --model llama3.1-8b \
    --num-users 100 \
    --duration 300 \
    --gpu-mem-gb 16 \
    --cpu-mem-gb 32 \
    --generation-mode realistic \
    --cache-dir /mnt/nvme \
    --seed 42 \
    --output results_realistic_production.json
```

What to look for: Compare the `end_to_end_latency_ms` from this test to the storage-only test. You should see a dramatic improvement. Also, check the `cache_hit_rate` and tier distribution (`gpu_entries` , `cpu_entries` , `nvme_entries`) to see how effectively your system is using the faster tiers.

Step 3: Discover Your System's Maximum User Load (QoS Mode)

This command enables the default **Quality of Service (QoS)** autoscaler. It finds the optimal number of concurrent users your hardware can support *while maintaining acceptable latency*. It starts with a low user count and adds more users until the system's storage latency indicates it is becoming saturated.

```
# Test 3: Autoscaling Discovery (QoS Mode)
# (Set --gpu-mem-gb to your available VRAM, or 0 if none)
python3 kv-cache.py \
    --model llama3.1-8b \
    --num-users 20 \
    --duration 300 \
    --gpu-mem-gb 16 \
    --cpu-mem-gb 32 \
    --enable-autoscaling \
    --autoscaler-mode qos \
    --generation-mode realistic \
    --cache-dir /mnt/nvme \
    --seed 42 \
    --output results_autoscaling_qos.json
```

What to look for: The output JSON will contain an `autoscaling_stats` section. The last entry in this list will show the final, stable user count your system settled on. This is your evidence-based maximum user load for a latency-sensitive production environment.

Step 4: Discover Your System's Peak Throughput (Capacity Mode)

This command uses the new **Capacity** autoscaler. Its goal is different: it ignores latency and aggressively adds users to find the absolute maximum I/O throughput (in tokens/sec) your storage hardware can sustain. This is the best way to measure the raw power of your drive.

```
# Test 4: Autoscaling Discovery (Capacity Mode)
python3 kv-cache.py \
    --model llama3.1-70b-instruct \
    --num-users 10 \
    --duration 180 \
    --gpu-mem-gb 0 \
    --cpu-mem-gb 32 \
    --enable-autoscaling \
    --autoscaler-mode capacity \
    --generation-mode none \
    --cache-dir /mnt/nvme \
    --seed 42 \
    --output results_autoscaling_capacity.json
```

What to look for: In the `autoscaling_stats` section, look for the `reason` field. The test finishes when it detects that throughput has stopped increasing. The final log will state `Peak capacity found`. The `peak_throughput` value associated with that step is the maximum performance of your storage device. Note the use of `--generation-mode none` to ensure the storage is the only bottleneck.

---

3. Hardware Requirements

To effectively run this benchmark and obtain meaningful results, your system should meet certain hardware specifications. The benchmark is flexible, but the quality of your results will depend on the hardware's capabilities, especially the storage subsystem. This is an enterprise storage test, and the recommendations reflect server-grade hardware.

Minimum Requirements

These specifications are sufficient to run the basic `storage-only` workload and validate the functionality of the benchmark with a low user count.

- **CPU:** 8+ Core Server-Grade CPU (e.g., AMD EPYC, Intel Xeon Bronze/Silver)
- **System RAM:** 32 GB ECC RAM
- **GPU:** Not required. The benchmark can run in CPU-only mode (`--gpu-mem-gb 0`).
- **Storage:** 256 GB+ of free space on a data center-class SATA/SAS SSD.
- **Operating System:** A modern Linux distribution (e.g., Ubuntu 22.04, RHEL 9) is required for best performance and compatibility.

Recommended Specifications

These specifications are recommended for running the full suite of tests, including the `realistic` multi-tier and `autoscale` workloads with a high user count. This configuration will provide a robust analysis of your system's ability to handle a production-level inference load.

- **CPU:** 32+ Core Server-Grade CPU (e.g., AMD EPYC 9354 "Genoa", Intel Xeon Gold/Platinum 4510+)
- **System RAM:** 128 GB ECC RAM or more. This allows for a significant CPU cache tier (`--cpu-mem-gb 64` or higher).
- **GPU:** An NVIDIA Data Center GPU (e.g., A100, H100) with 40GB+ of HBM. This is necessary to test the complete three-tier hierarchy at scale.
- **Storage:** 1 TB+ of free space on a high-performance, data center-class NVMe SSD (e.g., PCIe Gen4 or Gen5). The primary goal of this benchmark is to measure the performance of this tier.
- **Operating System:** A modern Linux distribution (e.g., Ubuntu 22.04, RHEL 9).

---

4. Automating the Benchmark with `kv-cache-wrapper.sh`

While you can run each test scenario manually using `kv-cache.py` , the repository includes a powerful wrapper script, `kv-cache-wrapper.sh` , to automate the entire process. This script is the recommended way to get a comprehensive performance profile of your system with minimal effort.

The wrapper script will:

1. **Automatically detect your hardware:** It checks for available GPU(s), total CPU RAM, and the best path for storage testing.
2. **Calculate optimal parameters:** It determines reasonable user counts and memory budgets based on your hardware to ensure the tests are meaningful but not destructive.

3. **Run a full suite of 9 tests:** It executes a series of pre-configured benchmarks to compare every possible tier configuration and stress test the system.
4. **Generate a summary report:** After all tests are complete, it prints a detailed comparison table, allowing you to easily see the performance trade-offs for your specific hardware.

How to Use the Wrapper

Running the script is simple. From your terminal, execute it directly. It's a good idea to pipe the output to a log file for later review.

```
# Run the full benchmark suite with default settings
./kv-cache-wrapper.sh | tee benchmark_summary.log

# Run the suite with a different model
./kv-cache-wrapper.sh -m llama3.1-70b-instruct

# Run only specific workloads, like the production and autoscale tests
./kv-cache-wrapper.sh -w production,autoscale
```

The script runs the following nine scenarios automatically:

- **Test 1: GPU Only:** A baseline for best-case latency, limited by VRAM.
- **Test 2: CPU Only:** A typical production setup using only system RAM.
- **Test 3: Storage Only:** Isolates the NVMe drive to measure its raw performance.
- **Test 4: GPU + CPU:** A two-tier configuration without storage spillover.
- **Test 5: CPU + Storage:** Simulates a budget-friendly setup with RAM and NVMe.
- **Test 6: GPU + CPU + Storage:** The full three-tier hierarchy for maximum capacity and performance.
- **Test 7: Storage Saturation:** A stress test to find the breaking point of your NVMe drive.
- **Test 8: Realistic Production:** A balanced, steady-state test mimicking a normal day.
- **Test 9: Autoscaling Discovery:** Automatically finds the maximum number of users your system can handle.

At the end of the ~30-minute run, the script will output a detailed report comparing the throughput, latency, and cache distribution for each scenario, giving you a clear, evidence-based picture of how your system performs.

---

5. A Look Under the Hood: How It Works

In the KV cache benchmark, a **user request** is an `InferenceRequest` data structure that simulates a single interaction, or "turn," with a Large Language Model.

- Each `InferenceRequest` object contains several key fields to model this interaction:
  - `context_tokens`: The number of tokens in the user's prompt. This directly determines the size of the initial KV cache that needs to be written to storage in the "prefill" phase.
  - `generate_tokens`: The number of tokens the model is asked to generate. In the "decode" phase, this influences how many times the existing KV cache is read from storage
  - `phase`: The type of I/O operation, which can be `PREFILL` (write-heavy), `DECODE` (read-heavy), or a combination of both (`PREFILL_DECODE`)
  - `cache_key`, `conversation_id`, and `turn_number`: These fields link requests to simulate multi-turn conversations, where the cache from a previous turn must be read to generate the next response.
- Cache hit categories (e.g., `'system'`, `'common'`, `'multi_turn'`, `'user'`) are determined by a `cache_type` hint that the `process_requests` function passes to the `access_cache` method. This categorization is provided by the caller at the time of access, so `InferenceRequest` remains agnostic.
- The benchmark measures two critical types of latency for each request:
  - **Storage I/O Latency**: This metric measures the time elapsed from the moment a cache operation (`access_cache` for reads or `allocate_cache` for writes) is invoked until it returns. Critically, this duration includes not only the hardware I/O time but also all user-space software overhead within those functions, such as the CPU-intensive process of serializing or deserializing NumPy arrays. It does *not* include time the request spent waiting in the main application queue.
  - **End-to-End Latency**: This is the total time the user experiences, measured from request creation (`submit_time`) to completion (`complete_time`). It is the sum of **Queue Wait Time** + **Storage I/O Latency** + **Token Generation Latency**.
- A `UserSimulator` generates a mix of these requests based on different user "personas" (e.g., 'chatbot', 'coding') to create a realistic workload with varied prompt sizes and response lengths

The benchmark uses these requests to simulate the two primary phases of inference, which have distinct I/O patterns:

1. **Initial Prefill (Turn 1):** For the first request in a conversation, the benchmark generates a NumPy array for the user's `context_tokens` and writes it to a storage tier using the `MultiTierCache.allocate_cache` function. This is a single, write-heavy operation.
2. **Subsequent Prefills (Turn > 1):** For the next turn in the same conversation, the process simulates loading the existing context before adding new information in a read-then-write pattern:
    - **Read Previous Context:** The `process_requests` loop first performs a **read** operation. It calls `self.cache.access_cache` on the cache key from the *previous* turn (e.g., `conversation-ID_turn_1`) to simulate loading the conversational history.
    - **Write New Context:** It then generates a new NumPy array for the *new* `context_tokens` of the current turn and performs a **write** operation by calling `self.cache.allocate_cache` with a new key (e.g., `conversation-ID_turn_2`)

How are the KV cache entries are stored on the XFS file system?

- a unique cache_key is generated for every request.
- The InferenceRequest class generates a key based on its context. For multiturn conversation its tied to a turn number.
- The key is then used to create a unique filepath, then the data is saved to that single file (per request).

```
  def post_init(self):

     if self.cache_key is None:

        if self.conversation_id:

           self.cachekey = f"{_self.conversationid}_turn{self.turn_number}"

        else:

           self.cachekey = f"{_self.user_id}_ctx"

class NVMeBackend(StorageBackend):

  def get_path(_self, key: str) -> Path:

     """Constructs the file path for a given cache key."""

     return self.basepath / f"{_key}.npy"

  def write(self, key: str, data: np.ndarray) -> StorageBackend.IOTiming:

     path = self.get_path(_key)

     with open(path, 'wb') as f:

        np.save(f, data, allow_pickle=False)
```

A. The Three-Tier Architecture: A Hierarchy of Speed

The benchmark's core is the `MultiTierCache` class, which implements a classic three-tier memory hierarchy. The goal is to keep the "hottest" (most frequently accessed) data in the fastest tier (GPU) and the "coldest" data in the slowest but largest tier (NVMe).

1. **Tier 1: GPU VRAM (`GPUMemoryBackend`):** The fastest tier. Data is stored as PyTorch or CuPy tensors for near-instant access. Capacity is extremely limited and expensive.
2. **Tier 2: CPU RAM (`CPUMemoryBackend`):** The "warm" tier. Data is stored as NumPy arrays in system memory. It's an order of magnitude slower than VRAM but much larger and cheaper.
3. **Tier 3: NVMe Storage (`NVMeBackend`):** The "cold" tier. Data is written to `.npy` files on disk. It offers massive capacity at the lowest cost but with the highest latency.

How Data Placement is Decided (`allocate_cache`):
When a new KV cache entry needs to be created (during the "prefill" phase), the benchmark follows a simple, top-down logic:

```
# From kv-cache.py, inside MultiTierCache.allocate_cache
with self.memory_lock:
    # Tier 1: GPU. Check if there's space in the GPU budget (with a 20% buffer).
    if 'gpu' in self.backends and self.gpu_memory_used + size_bytes < self.gpu_memory_limit * 0.8:
        self.gpu_memory_used += size_bytes
        allocated_tier = 'gpu'
    # Tier 2: CPU. Check if there's space in the CPU budget.
    elif self.cpu_memory_used + size_bytes < self.cpu_memory_limit * 0.8:
        self.cpu_memory_used += size_bytes
        allocated_tier = 'cpu'
    # Tier 3: NVMe. If no space in RAM, offload to disk.
    else:
        allocated_tier = 'nvme'
```

Real-World Implication: This logic simulates how a real inference server would operate. It prioritizes the fastest memory available. If you configure the benchmark with a small GPU and CPU memory budget, you are forcing data to spill over to the NVMe drive, allowing you to measure the performance penalty of that spillover.

B. Memory Clamps: The 80% Rule

You'll notice the `* 0.8` in the allocation logic. This is a crucial design choice. The benchmark intentionally leaves a **20% headroom** on both the GPU and CPU memory limits.

Why?
This prevents the system from running completely out of memory, which can cause crashes, operating system swapping (thrashing), or out-of-memory (OOM) errors. It ensures that there is always a small buffer available for system processes and other application needs.

Real-World Implication: This is a best practice in production systems. You never want to run your memory at 100% utilization. The 80% rule provides stability and ensures that performance remains predictable. When sizing your own hardware, you should apply a similar rule: if you calculate that you need 64 GB of RAM, you should provision at least 80 GB.

C. Latency Calculation: User Experience vs. Hardware Speed

The benchmark reports two different types of latency, and the distinction is critical.

```
+---------------------------------+
|  Application (kv-cache.py) [1] |
|   - Request Queue (piles up) --------> "Queue Wait" is the dominant latency component [4]
|   - Multiple Worker Threads    |
+---------------------------------+
             |
             | A single worker thread grabs one request.
             v
+---------------------------------+
|   Worker Thread & Sync I/O     |
|  Issues 1 x LARGE, BLOCKING    |
|   Write/Read (e.g., 1 GB)      |
|                                |
|  [SUBMISSION QD = 1]          | --------> The thread BLOCKS and WAITS for completion.
+---------------------------------+
             |
             | OS receives the single large request.
             v
+---------------------------------+
|          Kernel / OS           |
|    Performs "I/O Splitting"    | --------> Splits 1 large I/O into hundreds of small ones.
+---------------------------------+
             |
             | Drive receives hundreds of small requests.
             v
+---------------------------------+
|     NVMe Storage Device        |
|  [DEVICE QD > 700] [3]           | --------> The drive is heavily utilized in a short burst.
|   Processes many small I/Os    |
|       in parallel            |
+---------------------------------+
```

1. **Storage I/O Latency:** This is the pure hardware time. It measures the time taken for a read or write operation to complete on a specific tier, **excluding any queue wait time.** It is accumulated within the `process_requests` loop every time `self.cache.access_cache` or `self.cache.allocate_cache` is called.
2. **End-to-End Latency:** This is the total time the user waits. It is measured from the moment a request is created (`submit_time`) to the moment it is finished (`complete_time`). It is the sum of **Queue Wait Time + Storage I/O Latency + Generation Latency.**

Real-World Implication:

- **Storage I/O Latency** tells you how good your hardware is. A low number means your drive is fast.
- **End-to-End Latency** tells you how good your system architecture is. A high number, even with a fast drive, indicates a bottleneck elsewhere—most commonly, in the request queue. As seen in the provided logs, the queue wait time can be orders of magnitude larger than the storage latency, proving that the system is overloaded.

D. Validating Latency with Block Tracing: Application vs. Hardware

As discussed in the previous section, the total **End-to-End Latency** is the sum of *Queue Wait Time* and *Storage I/O Latency*. The analysis below focuses on dissecting the *Storage I/O Latency* component, as this is where a crucial software bottleneck is revealed.

A common and important question is why the benchmark's "Storage I/O Latency" can be seconds long, even on a high-performance NVMe drive, while low-level tools like `btrace` show the drive is responding in milliseconds. This discrepancy is not an error; it is a key finding that reveals a crucial software bottleneck.

The two tools are measuring latency at different layers of the system:

1. **Application-Level I/O Latency (The Benchmark's Metric):** This is the total time spent inside the `NVMeBackend.read()` or `write()` methods in Python. This includes not only the time waiting for the disk, but also all associated software overhead, most notably the CPU-intensive process of serializing (saving) or deserializing (loading) the Python data structures (NumPy arrays) to and from a binary format on disk.
2. **Hardware-Level I/O Latency (`btrace`'s Metric):** This is the pure hardware time. It measures the time from when an I/O request hits the Linux block layer until the physical NVMe drive signals that the operation is complete. This is the true speed of your storage device.

Case Study: Analyzing the Discrepancy with Real Data

Let's examine the results from a real test run to see this in action.

- **From the Benchmark Log (`mlperf_log_run4.txt`):**
  The benchmark reports a P95 NVMe read latency of **12.39 seconds**.

  ```
  ### TIER-SPECIFIC LATENCIES ###
    NVME Read P95: 12390.15 ms
  ```
- **From the Block Trace Log (`btrace_analysis_btrace_read.txt`):**
  In contrast, a `btrace` analysis of the same workload shows the P95 hardware read latency was only **9.74 milliseconds**.

  ```
  D2C Latency Analysis: ... Latency (ms) 9.74
  ```

The Analysis:

The massive difference between these two numbers exposes the software overhead.

Metric	Source	Time
Total Application Latency	Benchmark Log	**12,390 ms**
Actual Hardware Latency	`btrace` Log	**~10 ms**
Software Overhead (CPU Serialization)	(Difference)	**~12,380 ms**

This clearly shows that for a P95 read operation, **over 99.9% of the time was spent in the CPU-bound `numpy.load()` function**, deserializing the data. The physical drive responded in under 10 milliseconds.

Conclusion: The `btrace` logs confirm the storage hardware is not the problem. The benchmark is correctly revealing a significant software bottleneck in the Python-based I/O path. A real-world, high-performance inference engine written in C++ or using technologies like GPUDirect Storage would aim to minimize or eliminate this CPU serialization step, resulting in application latency much closer to the hardware latency shown in `btrace`. This is a key finding of the benchmark: it successfully models not just the storage hardware's performance, but also the overhead of the software stack used to access it.

E. So, How Should You Interpret the Latency Numbers?

Given the different layers of latency, here is a simple guide to interpreting the results:

- **Use `End-to-End Latency` to judge User Experience.** This is the total time a user has to wait for a response. If this number exceeds your Service Level Agreement (SLA), your system is too slow for its workload, regardless of the reason.

- **Use `Queue Wait Time` to diagnose Overload.** If this number is high (or makes up a large portion of the End-to-End Latency), it is a clear sign that your system is receiving requests faster than it can process them. The bottleneck is system capacity.
- **Use `Storage I/O Latency` to evaluate the Application's I/O Path.** This number tells you the performance of your Python storage backend. If this number is high, it indicates a bottleneck in the software layer (like CPU serialization), as demonstrated in the case study above.
- **Use `btrace` (Hardware Latency) to evaluate the Physical Drive.** This number tells you the true speed of your NVMe device. If this number is low, your storage hardware is performing well.

In short, `btrace` checks the disk, `Storage I/O Latency` checks the application's I/O efficiency, `Queue Wait Time` checks for system overload, and `End-to-End Latency` checks the final user experience.

F. QoS Classes: Prioritizing Users

Not all inference requests are created equal. A user interacting with a chatbot needs an instant response, while a batch job summarizing a document can wait. The benchmark models this with three Quality of Service (QoS) levels defined in the `QoSLevel` enum.

G. The MLPerf Storage Submission: Finding the Breaking Point

The "MLPerf Storage" tests included in the wrapper script are designed to do one thing: find the absolute performance limit of the system by intentionally overloading it. When looking at the results, it's common to see extremely high latency numbers, which might seem alarming. However, in the context of a benchmark, this is not only expected, it is a sign of a successful test.

This state, often called "thrashing," is when the system is receiving requests so much faster than it can process them that it spends most of its time managing the backlog. This is the most demanding scenario for a storage subsystem.

Case Study: Interpreting a "Thrashing" Result

Let's analyze the provided results for the 8B model submission:

```
End-to-end latency: mean 317.96s, P50 322.10s, P95 635.48s
Approximate mean queue wait: 274.08s
Storage I/O latency: mean 37.04s, P95 138.50s
Potential bottlenecks:
  - Queue wait dominates (~274.08s mean).
```

The Analysis:

1. **The System is Overloaded:** The most telling metric is the `Approximate mean queue wait` of **274 seconds**. This means that, on average, a request spent over 4.5 minutes waiting in a queue before the system even began to process it.
2. **The Bottleneck is System Capacity:** The fact that queue wait time accounts for ~86% of the total end-to-end latency (274s out of 318s) is a definitive sign that the system as a whole cannot keep up with the request rate.
3. **The Storage is Under Extreme Stress:** Even after the long wait, the P95 `Storage I/O latency` is over two minutes (138.5s). As established previously, this is mostly due to application-level overhead, but it demonstrates the immense pressure on the I/O path. The system is desperately reading and writing from the NVMe drive to serve the KV cache for many concurrent users.

Why This is a Good Benchmark Result:

This is a valuable result precisely *because* it pushed the system to failure.

- **It finds the true bottleneck:** The test proves that under heavy load, the primary bottleneck isn't just the disk, but the system's overall capacity to handle concurrent requests, leading to massive queue times.
- **It validates the storage:** Despite the system thrashing, the storage subsystem continued to operate and serve terabytes of I/O without failing. This is the goal of the MLPerf Storage test: to certify that the storage solution is robust enough to handle a worst-case "denial-of-service" style workload.
- **It measures maximum throughput:** The reported `312.2 tok/s` is the throughput the system could sustain while being completely saturated. This represents the performance floor under maximum stress.

In conclusion, the MLPerf submission is not measuring performance under ideal conditions. It is a stress test designed to find the breaking point, and the resulting high latency numbers are a clear and useful indicator of where that breaking point is.

```python
# From kv-cache.py
class QoSLevel(Enum):
    INTERACTIVE = "interactive" # Highest priority, for real-time applications (e.g., chatbot UI).
    RESPONSIVE = "responsive"   # High priority, for near real-time tasks.
    BATCH = "batch"             # Low priority, for offline processing.
```

Each QoS level has a Service Level Agreement (SLA) with a target P95 latency. The benchmark uses a `PriorityQueue` to ensure that `INTERACTIVE` requests are always processed before `BATCH` requests, simulating how a real production scheduler would work.

Real-World Implication: This feature allows you to test whether your hardware can meet the strict latency demands of high-priority users while still processing a background load of low-priority tasks.

F. Autoscaling: Finding Your System's True Limit

The `WorkloadAutoscaler` is perhaps the most powerful feature of the benchmark. Instead of guessing the number of users or throughput your system can handle, it finds it automatically using one of two modes, selectable with the `--autoscaler-mode` flag.

Mode 1: `qos` (Quality of Service)

This is the default mode, designed for system architects tuning a **production environment**. Its goal is to find the maximum number of users the system can support while keeping latency low to ensure a good user experience.

How it works:

1. The `StorageMonitor` periodically collects key performance indicators (KPIs), primarily P95 read latency from the storage tiers.
2. It uses these KPIs to calculate a `saturation` score from 0.0 (idle) to 1.0 (fully saturated). A key heuristic is rising latency.
3. The `WorkloadAutoscaler` compares this saturation score to a target (defaulting to `0.8`, or 80%).
   - If saturation is too low, it increases the number of simulated users.
   - If saturation is too high, it decreases the number of users.
   - It includes a "cooldown" period after a scale-down to allow the system to stabilize.

Real-World Implication: This mode allows you to provision your hardware with confidence. By running this test, you can determine the maximum safe user load for your specific server configuration and use that number to set the limits in your production load balancer, ensuring good performance.

Mode 2: `capacity` (Peak Throughput)

This mode is designed for hardware vendors and performance engineers who want to find the **absolute peak throughput** of a storage device, ignoring user-facing latency.

How it works:

1. The autoscaler starts with a low user count.
2. It aggressively doubles, then increases the user count by 1.5x in stages, monitoring the total `tokens/sec` throughput at each stage.
3. When it detects that adding more users causes the throughput to *decrease* (meaning the point of diminishing returns has been passed), the test concludes.
4. The result is the highest throughput measured before the drop.

Real-World Implication: This is the purest test of raw hardware performance. By combining it with `--generation-mode none`, you can remove all other bottlenecks and measure the maximum I/O your storage can deliver. This is invaluable for comparing the performance of different SSDs in an "apples-to-apples" test.

G. RAG Workflow: Simulating Modern Workloads

Retrieval-Augmented Generation (RAG) is a popular technique where an LLM's context is "augmented" with relevant documents. This creates a unique I/O pattern that the benchmark simulates with the `RAGDocumentManager`.

How it works:

1. **Ingestion (`ingest_document`):** The benchmark simulates the "ingestion" of large documents by splitting them into chunks and pre-calculating and storing the KV cache for each chunk across the three-tier hierarchy.
2. **Retrieval (`retrieve_chunks`):** When a RAG query is simulated, the benchmark retrieves the `top_k` most relevant chunks. This simulates a vector database lookup.
3. **Inference:** The retrieved chunks are then used as the context for the LLM, which involves reading the pre-calculated KV cache for each chunk from storage.

Real-World Implication: RAG workloads place immense stress on the storage system because they involve loading very large contexts (many document chunks) into memory at the start of a request. This feature allows you to test whether your storage can handle the bursty, high-throughput read demands of a RAG-based application.

H. Generation Mode: Simulating GPU Backpressure

A storage benchmark for LLM inference would be incomplete if it only measured I/O. In a real system, the GPU is constantly performing computations to generate the next token. This computation time creates **backpressure** on the I/O subsystem. The benchmark cannot make another I/O request until the GPU is finished with its current work. Without simulating this, the benchmark would flood the storage with requests at an unrealistic rate.

The `--generation-mode` flag controls this simulation by adding a small `time.sleep()` for each token generated.

```
# From kv-cache.py
class GenerationMode(Enum):
    NONE = "none"           # Pure storage benchmark. No simulated sleep. Latency is 100% I/O.
    FAST = "fast"           # Simulates a very fast GPU (2ms/token) to model some backpressure.
    REALISTIC = "realistic" # Simulates a realistic GPU (30ms/token) for end-to-end latency analysis.

GENERATION_TIMING = {
    GenerationMode.NONE: 0.0,
    GenerationMode.FAST: 0.002,
    GenerationMode.REALISTIC: 0.030,
}
```

How These Values Were Derived:

- `none` **(0 ms/token):** This is for pure storage hardware validation. It removes all simulated GPU processing time to measure the absolute maximum I/O throughput the storage can handle. This mode is useful for finding the raw performance of a drive but does not represent a real-world LLM serving scenario.
- `realistic` **(30 ms/token):** This is the most important mode for system-level testing and is **required for MLPerf submissions**. The 30ms value was derived from empirical measurements of modern data center GPUs (like the NVIDIA A100 or H100) running medium-sized models (7B-8B parameters). This latency corresponds to a generation speed of approximately **33 tokens per second**, which is a standard and widely accepted performance figure for these models in production. Using this mode ensures the benchmark paces its I/O requests at a rate that a real GPU could sustain.
- `fast` **(2 ms/token):** This mode simulates a very high-performance or next-generation accelerator, capable of generating **500 tokens per second**. It is useful for modeling "what-if" scenarios where the GPU is so fast that it is almost never the bottleneck, thereby placing maximum stress on the memory and storage hierarchy.

Real-World Implication: For any test that aims to measure system-level performance (like the `realistic` or `autoscale` workloads), you must use `--generation-mode realistic`. Failure to do so will result in misleadingly high throughput numbers and will not accurately represent the performance of a balanced, production-ready system.

I. Shared System Prompts and Prefix Reuse

Most chat products send the same “system prompt” (for example, *“You are a helpful assistant.”*) before every user message. In real deployments the platform tries to reuse that prompt instead of regenerating it every time:

1. The first conversation runs the full prefill step and stores the prompt’s KV cache in fast memory (GPU or CPU).
2. Later conversations look up that stored block. If it is still around, they read it and skip the extra work. If it has been evicted, they rebuild it and store it again.

The benchmark copies that pattern with three simple pieces:

- **Detect:** `PrefixMatcher` pretends ~20 % of requests start with one of three common prompts. It hashes the text so everyone shares the same key (`kv_system_<hash>`).
- **Count reuse attempts:** `PrefixCacheManager` records how often the matcher sees the prompt. The `system_prompt_reuse` counter therefore means “we spotted the pattern,” even if the cache entry is missing.
- **Count real hits:** `MultiTierCache.access_cache` tries to read the shared key. If the block exists, `system_prompt_hits` increments. If not, the request falls back to a normal prefill.

In the summary you will see both numbers. A high reuse count with few hits simply says the prompt was detected but the stored copy had already been evicted, just like what operators watch for in production.

---

6. Current Work: Validating Simulation Accuracy with vLLM

The primary goal of `kv-cache.py` is to provide a reliable *simulation* of a multi-tiered KV Cache system. But how do we know the simulation is accurate? We must validate it against a real-world, high-performance inference engine. For this, we use **vLLM**, a state-of-the-art LLM serving library.

Our validation process is divided into two essential steps:

1. **Baseline Validation (GPU-Only):** First, we establish a performance baseline by running both `kv-cache.py` and vLLM in a GPU-only configuration. This test ensures that the core token generation logic of the simulator is accurate when no memory offloading occurs.
2. **Offloading Validation (GPU + CPU):** Second, we validate the primary feature of the benchmark: cache offloading. We configure both tools with limited GPU memory to force the KV cache to spill into CPU RAM, and then we compare the performance impact.

The pass/fail criterion for both steps is the same: the **tokens per second** reported by `kv-cache.py` should be within **±5%** of the tokens per second reported by vLLM's benchmark tool.

Step 1: Baseline Validation (GPU-Only)

In this step, we configure both tools to use a small model and a low user count, ensuring all KV cache data remains within the GPU's VRAM. This isolates the performance of the GPU and the core generation loop.

A. `kv-cache.py` Command (GPU-Only):

We run the benchmark with a high GPU memory budget and zero CPU/NVMe budget. This forces all allocations into the `GPUMemoryBackend`. Using a fixed seed ensures the workload is identical for comparison.

```
# Validation Step 1: Run kv-cache.py in GPU-only mode
python3 kv-cache.py \
    --model llama3.1-8b \
    --num-users 10 \
    --duration 120 \
    --gpu-mem-gb 24 \
    --cpu-mem-gb 0 \
    --generation-mode deterministic \
    --seed 42 \
    --output validation_kv_cache_gpu_only.json
```

B. vLLM Command (GPU-Only):

We run vLLM's offline benchmark without providing any swap space. This ensures vLLM does not offload any cache data to the CPU. The `--num-prompts` should match the `--num-users` from the `kv-cache.py` command. If you haven't already, you can install vLLM with pip:

```
pip install vllm
```

Now, run the vLLM benchmark:

```
# Validation Step 1: Run vLLM benchmark in GPU-only mode
python3 -m vllm.entrypoints.cli.main bench throughput \
    --model meta-llama/Llama-3.1-8B \
    --dataset-name random \
    --num-prompts 10 \
    --input-len 1024 \
    --output-len 1024
```

C. Compare Results:

Compare the `total_tokens_per_sec` from `validation_kv_cache_gpu_only.json` with the `total tokens/s` from the vLLM output. They should be within 5% of each other.

Step 2: Offloading Validation (GPU + CPU)

Here, we validate the simulator's main purpose: measuring the performance impact of cache offloading. We reduce the available GPU memory to force both `kv-cache.py` and vLLM to use CPU RAM as a secondary cache tier.

A. `kv-cache.py` Command (GPU + CPU):

We reduce the GPU memory budget to force allocations to spill over to the `CPUMemoryBackend`.

```
# Validation Step 2: Run kv-cache.py with GPU-to-CPU offloading
python3 kv-cache.py \
    --model llama3.1-8b \
    --num-users 20 \
    --duration 120 \
    --gpu-mem-gb 8 \
    --cpu-mem-gb 32 \
    --generation-mode deterministic \
    --seed 42 \
    --output validation_kv_cache_offload.json
```

B. vLLM Command (GPU + CPU):

We use the `--swap-space` argument to tell vLLM to allocate a KV cache in CPU RAM. The user count is increased to ensure this space is utilized.

```
# Validation Step 2: Run vLLM benchmark with GPU-to-CPU offloading
python3 -m vllm.entrypoints.cli.main bench throughput \
    --model meta-llama/Llama-3.1-8B \
    --dataset-name random \
    --num-prompts 20 \
    --input-len 1024 \
    --output-len 1024 \
    --swap-space 16
```

C. Compare Results:

Again, compare the `total_tokens_per_sec` from `validation_kv_cache_offload.json` with the `total tokens/s` from the vLLM output. A successful validation will see the results within the ±5% margin, confirming that `kv-cache.py` accurately models the performance penalty of offloading.

Hardware & Software Requirements for Validation

To run this validation, you will need:

- **Hardware:** An NVIDIA GPU with at least 16 GB of VRAM and Compute Capability 7.0+ (e.g., V100, T4, A100, RTX 30/40 series).
- **Environment:** A Linux environment (or WSL 2 on Windows).
- **Software:** Python 3.10+, PyTorch, and vLLM installed (`pip install vllm`).

---

7. MLPerf v3.0 Submission Guidelines

For submitting official results to the MLPerf v3.0 benchmark, it is critical to use a standardized, repeatable methodology that isolates the component being tested. When evaluating a storage device's capability for KV cache offloading, the goal is to measure the performance of the storage subsystem under a consistent and saturating load, even on systems without a high-end GPU.

Recommended Invocations for Storage Submission

Two primary scenarios should be submitted to give a comprehensive view of storage performance: a standard test with a medium-sized model (Llama 3.1 8B) and a high-stress test with a large model (Llama 3.1 70B).

Standard Submission: `llama3.1-8b`

This workload provides a baseline for storage performance under typical conditions. A fixed seed is required to ensure the workload is identical for all submissions, enabling fair and reproducible comparisons.

```
# MLPerf v3.0 Recommended Invocation: Storage Saturation Test (8B Model)
python3 kv-cache.py \
    --model llama3.1-8b \
    --num-users 150 \
    --duration 600 \
    --gpu-mem-gb 0 \
    --cpu-mem-gb 2 \
    --generation-mode realistic \
    --performance-profile throughput \
    --seed 42 \
    --output mlperf_v3_storage_submission_8b.json
```

Large Model Submission: `llama3.1-70b-instruct`

This workload tests the storage's ability to handle a much heavier load, as the KV cache for a 70B model is significantly larger. The user count is reduced to reflect the increased memory pressure per user.

```
# MLPerf v3.0 Recommended Invocation: Storage Saturation Test (70B Model)
python3 kv-cache.py \
    --model llama3.1-70b-instruct \
    --num-users 40 \
    --duration 600 \
    --gpu-mem-gb 0 \
    --cpu-mem-gb 4 \
    --generation-mode realistic \
    --performance-profile throughput \
    --seed 42 \
    --output mlperf_v3_storage_submission_70b.json
```

Key Parameters Explained:

- `--num-users 150` : A high, fixed user count is used to ensure the storage device is placed under significant and continuous load.
- `--duration 600` : A 10-minute duration ensures the benchmark reaches a stable, steady-state performance level, which is a standard requirement for MLPerf results.
- `--gpu-mem-gb 0` : **This is the critical parameter for a storage-focused test.** It ensures the benchmark does not allocate any GPU memory, making it suitable for systems without a GPU or for isolating storage performance.
- `--cpu-mem-gb 2` : This small memory budget is intentionally chosen to be insufficient for the user load, forcing the system to bypass this faster tier and offload almost all KV cache data directly to the NVMe storage.
- `--generation-mode realistic` : This is essential for a valid submission. It adds a 30ms emulated sleep for each token generated, accurately simulating the backpressure from a real GPU's computation time. Without this, the benchmark would incorrectly measure storage performance in an unrealistic, I/O-only scenario.
- `--performance-profile throughput` : This new parameter is crucial for official submissions. It instructs the benchmark to use **throughput (tokens/second) as the sole pass/fail metric**, ignoring latency. This is because the high user count and low memory budget are *designed* to cause high latency to saturate the storage. This profile ensures the benchmark correctly evaluates the storage device's ability to sustain a high data rate under stress, which is the true goal of this test.
- `--seed 42` : **This parameter is mandatory for a valid submission.** It ensures that the pseudo-random workload (user request timings, context lengths, etc.) is identical across all test runs and systems. This removes workload variance as a factor and guarantees a true "apples-to-apples" comparison of hardware performance. The final report will include the seed used.

Interpreting Throughput: System vs. Storage (Read Amplification)

When you run the benchmark with the `throughput` profile, the summary report presents two different throughput numbers that can differ significantly. Understanding this difference is key to correctly interpreting the results.

1. **System Throughput (`total_tokens_per_sec`):** This is the "Overall Performance" metric. It represents the end-to-end throughput of the entire system from the user's perspective: the number of new tokens generated per second across all users. It is a measure of the system's generative capacity.
2. **Storage Throughput (`nvme_throughput`):** This is the "Storage Performance Assessment" metric. It represents the raw I/O performance of the NVMe tier, measuring how many tokens' worth of KV cache data are read from or written to the storage device per second.

Why Are They So Different? The Concept of Read Amplification

The Storage Throughput is often an order of magnitude higher than the System Throughput. This is not a bug; it is a fundamental characteristic of LLM inference called **Read Amplification**.

During the "decode" phase, to generate a single new token, the model must read the *entire KV cache for all preceding tokens in the conversation*.

- **Example:** A user has a context of 1000 tokens. To generate the 1001st token, the system must read the KV cache for all 1000 previous tokens from storage.

- **System Tokens Generated:** 1
- **Storage Tokens Read:** 1000

This creates a massive amplification effect where a small amount of user-facing work (generating one token) triggers a large amount of backend I/O (reading the entire history). This is precisely the behavior this benchmark is designed to measure, as it is the primary source of stress on the storage subsystem in a real-world KV cache offloading scenario.

Code Snippets

1. System Throughput Calculation:
This metric is calculated in the `_calculate_stats` method and is based on the number of new tokens generated.

```
# From IntegratedBenchmark._calculate_stats in kv-cache.py
total_tokens_generated = self.stats['tokens_generated']
if duration > 0:
    self.stats['total_tokens_per_sec'] = total_tokens_generated / duration
```

2. Storage Throughput Calculation:
This metric is calculated in the `_evaluate_storage_performance` method and is based on the `nvme_tokens_processed` counter, which tracks all I/O to the NVMe tier.

```
# From MultiTierCache._evaluate_storage_performance in kv-cache.py
nvme_tokens = self.stats.get('nvme_tokens_processed', 0)
if duration > 0:
    nvme_throughput = nvme_tokens / duration
```

3. How Storage Tokens are Counted:
The `nvme_tokens_processed` counter is incremented during both writes (`allocate_cache`) and reads (`access_cache`) that involve the NVMe tier.

Writing to NVMe (Prefill):

```
# From MultiTierCache.allocate_cache in kv-cache.py
if allocated_tier == 'nvme':
    # For throughput calculation, track tokens written to NVMe
    if self.performance_profile == 'throughput':
        self.stats['nvme_tokens_processed'] += num_tokens
```

Reading from NVMe (Decode):

```
# From MultiTierCache.access_cache in kv-cache.py
elif key in self.nvme_entries:
    # ...
    # For throughput calculation, track tokens read from NVMe
    if self.performance_profile == 'throughput':
        entry_size = self.nvme_entries[key]['size']
        num_tokens = entry_size // self.model_config.kv_cache_size_per_token
        self.stats['nvme_tokens_processed'] += num_tokens
```

By understanding read amplification, you can correctly interpret a high Storage Throughput not as an error, but as an accurate measurement of the intense I/O load the storage device is successfully handling.

What About RAG Workloads?

The benchmark includes a Retrieval-Augmented Generation (RAG) simulation mode (`--enable-rag`), which models workloads that inject large documents into the context. This creates a very large, write-heavy prefill phase and is an excellent way to stress-test a storage device's ability to handle bursty I/O.

However, for an official MLPerf submission, **it is recommended *not* to use the RAG workload.** The standard conversational workload provides a more consistent and repeatable I/O profile that is better suited for "apples-to-apples" comparisons between different storage solutions.

The RAG workload can be considered an optional, supplementary test. Vendors are encouraged to run it and report the results separately to showcase performance on this specific, demanding use case, but it should not replace the standard Storage Saturation test for the official submission.

Why Not Use Autoscaling for Submission?

The autoscaling feature (`--enable-autoscaling`) is an invaluable tool for system architects to discover the maximum user capacity of a *specific, balanced hardware configuration*. It is designed for system tuning and capacity planning, not for standardized component benchmarking.

For an official MLPerf submission focused on storage, a fixed-load test is superior for two reasons:

1. **Repeatability:** A fixed user count ensures that every test run applies the exact same load, leading to highly repeatable and consistent results. Autoscaling, by its nature, adjusts the load based on system performance, which can introduce variability between runs.
2. **Comparability:** The goal of MLPerf is to compare components on an "apples-to-apples" basis. By using a standardized, high-load command, we can directly compare the performance of different storage devices under the exact same conditions. Autoscaling would result in different final user counts for different systems, making direct comparison of the storage's throughput and latency difficult.

Therefore, the **Storage Saturation** test with a fixed, high user count is the correct methodology for generating official, comparable MLPerf v3.0 results for KV cache storage offloading.

---

8. Known Limitations and Future Work

This benchmark is a sophisticated tool for simulating KV cache offloading, but like any simulation, it has limitations. Understanding these is key to interpreting the results correctly and identifying areas for future improvement.

- **NumPy Serialization Overhead:** The `NVMeBackend` uses `numpy.save()` and `numpy.load()` to write and read cache entries to disk. While efficient, this process involves CPU-bound serialization and deserialization steps. A real-world inference engine might use more advanced techniques like GPUDirect Storage to move data directly from the GPU to NVMe, bypassing the CPU and avoiding this overhead. Therefore, the measured NVMe latency in this benchmark may be slightly higher than what is achievable with a fully optimized, custom storage pipeline.
- **Abstracted Storage Backends:** The benchmark currently provides a file-based `NVMeBackend` . It does not include built-in backends for other storage systems like object storage (e.g., S3), network file systems (NFS), or in-memory databases (e.g., Redis). While the `StorageBackend` class is extensible, testing these other systems would require implementing new backend classes.
- **Single-Node Architecture:** The simulation runs on a single machine, modeling multiple users through threading. It does not account for network latency or bandwidth, which would be a significant factor in a distributed inference environment where the KV cache might be stored on a separate, networked storage server.
- **Simulated GPU Backpressure:** The `--generation-mode` flag uses `time.sleep()` to emulate the time a GPU would spend on computation. This is a fixed-time approximation. It does not model the complex, dynamic nature of real GPU workloads, including variations in kernel execution times or PCIe bus contention between compute and I/O operations.
- **Simplified Eviction Policy:** The benchmark employs a straightforward Least Recently Used (LRU) policy for evicting old conversations when memory limits are reached. Production inference servers may use more complex eviction algorithms (e.g., Least Frequently Used, size-based eviction) to optimize cache hit rates.

An Invitation to Collaborate

This benchmark is an open-source effort driven by the MLPerf Storage Working Group. We welcome contributions from the community to help address these limitations and make the tool even more representative of real-world inference workloads.

If you are an expert in storage systems, GPU programming, or LLM inference and are interested in contributing, please consider getting involved. Areas where we would particularly value collaboration include:

- Developing new storage backends (e.g., for object storage or RDMA).
- Integrating more sophisticated GPU simulation models.
- Implementing alternative cache eviction policies.
- Expanding the benchmark to a distributed, multi-node architecture.

By working together, we can create a world-class, open standard for evaluating storage performance for AI.

H. How to Calculate Memory Requirements

A common point of confusion is the memory consumption of the benchmark, especially when testing large models like `llama3.1-70b-instruct`. It's natural to see a 70B model and expect memory usage to be in the hundreds of gigabytes, yet the benchmark process might only consume 15-20 GB of RAM.

This discrepancy arises because **the benchmark only simulates the I/O for the Key-Value (KV) cache; it does not load the model's actual weights.**

The primary goal of this tool is to measure the performance of your memory and storage subsystems under the specific I/O patterns generated by moving the KV cache between tiers. The 140GB+ of the model's weights are assumed to be static and already loaded in GPU VRAM. The benchmark focuses on the dynamic part: the KV cache, which is generated on-the-fly for each user.

The KV Cache Size Formula

The size of the KV cache for a single token can be calculated using the model's architectural parameters. The formula is:
Bytes per Token = `num_layers` **× 2 ×** `kv_heads` **× (** `hidden_dim` **/** `num_heads` **) ×** `bytes_per_dtype`

Where:

- `num_layers`: The number of transformer layers in the model.
- `2`: Represents the two components of the cache: the Key (K) and the Value (V).
- `kv_heads`: The number of attention heads for Keys/Values. For models using Grouped-Query Attention (GQA), this is smaller than `num_heads`.
- `hidden_dim / num_heads`: This calculates the dimension of a single attention head.
- `bytes_per_dtype`: The number of bytes for the data type (e.g., 2 for `float16`).

Calculation for Each Model

Here is the full calculation for each model defined in `kv-cache.py`:

- **`tiny-1b`**:
  - `32 × 2 × 4 × (1024 / 8) × 2` = **24,576 Bytes/Token** (~0.02 MB/Token)
- **`mistral-7b`**:
  - `32 × 2 × 8 × (4096 / 32) × 2` = **131,072 Bytes/Token** (~0.13 MB/Token)
- **`llama2-7b`** (Uses Multi-Head Attention, so `kv_heads` = `num_heads`):
  - `32 × 2 × 32 × (4096 / 32) × 2` = **524,288 Bytes/Token** (~0.50 MB/Token)
- **`llama3.1-8b`**:
  - `32 × 2 × 8 × (4096 / 32) × 2` = **131,072 Bytes/Token** (~0.13 MB/Token)
- **`llama3.1-70b-instruct`**:
  - `80 × 2 × 8 × (8192 / 64) × 2` = **327,680 Bytes/Token** (~0.31 MB/Token)

Memory per User for an 8K Context

Using these values, we can create a table showing the total KV cache size for a single user with a context of 8,192 tokens. This is crucial for capacity planning.

Model	Bytes per Token	Cache Size for 8,192 Tokens
`tiny-1b`	24,576	~192 MB
`mistral-7b`	131,072	~1,024 MB (1 GB)
`llama2-7b`	524,288	~4,096 MB (4 GB)
`llama3.1-8b`	131,072	~1,024 MB (1 GB)
`llama3.1-70b-instruct`	327,680	~2,560 MB (2.5 GB)

This table clearly illustrates the memory pressure. If you are running the `llama3.1-70b-instruct` model with 40 users, the total active KV cache size the benchmark needs to manage is `40 users * 2.5 GB/user = 100 GB`. If you only provide 4 GB of CPU RAM (`--cpu-mem-gb 4`), the benchmark will correctly offload the other ~96 GB to your NVMe drive, allowing you to measure the performance of your storage under that specific, heavy load.

---

9. Smoke Test: Quick Validation Suite

This section provides a collection of key benchmark invocations that can be used as a "smoke test" to quickly validate different aspects of your system's performance. Each test is designed to isolate a specific component or behavior. For all commands, it is assumed the cache directory is `/mnt/nvme`.

Test 1: Storage-Only Saturation

Purpose: Establishes the baseline performance of your storage device by forcing all I/O to it. This is the best way to measure your drive's raw throughput.

```
python3 kv-cache.py \
    --model llama3.1-8b \
    --num-users 50 \
    --duration 180 \
    --gpu-mem-gb 0 \
    --cpu-mem-gb 0.5 \
    --generation-mode realistic \
    --cache-dir /mnt/nvme \
    --seed 42 \
    --output results_storage_only.json
```

Test 2: Realistic Three-Tier Workload

Purpose: Simulates a balanced, production-level environment using GPU, CPU, and NVMe tiers. Use this to measure end-to-end latency in a typical setup.

```
python3 kv-cache.py \
    --model llama3.1-8b \
    --num-users 100 \
    --duration 300 \
    --gpu-mem-gb 16 \
    --cpu-mem-gb 32 \
    --generation-mode realistic \
    --cache-dir /mnt/nvme \
    --seed 42 \
    --output results_realistic_production.json
```

Test 3: Autoscaling for Max Users (QoS Mode)

Purpose: This is the key command for sizing your production environment. It automatically discovers the maximum number of concurrent users your system can support while maintaining a low-latency user experience (Quality of Service).

```
python3 kv-cache.py \
    --model llama3.1-8b \
    --num-users 20 \
    --duration 300 \
    --gpu-mem-gb 16 \
    --cpu-mem-gb 32 \
    --enable-autoscaling \
    --autoscaler-mode qos \
    --generation-mode realistic \
    --cache-dir /mnt/nvme \
    --seed 42 \
    --output results_autoscaling_qos.json
```

Test 4: Autoscaling for Peak Throughput (Capacity Mode)

Purpose: Ignores latency to find the absolute maximum I/O throughput (tokens/sec) your storage hardware can sustain. This is the ultimate test of your drive's raw power.

```
python3 kv-cache.py \
    --model llama3.1-70b-instruct \
    --num-users 10 \
    --duration 180 \
    --gpu-mem-gb 0 \
    --cpu-mem-gb 32 \
    --enable-autoscaling \
    --autoscaler-mode capacity \
    --generation-mode none \
    --cache-dir /mnt/nvme \
    --seed 42 \
    --output results_autoscaling_capacity.json
```

Test 5: MLPerf Storage Submission (8B Model)

Purpose: A standardized, high-load stress test designed to saturate the storage device and measure its sustained throughput for an official MLPerf submission.

```
python3 kv-cache.py \
    --model llama3.1-8b \
    --num-users 150 \
    --duration 600 \
    --gpu-mem-gb 0 \
    --cpu-mem-gb 2 \
    --generation-mode realistic \
    --performance-profile throughput \
    --seed 42 \
    --output mlperf_v3_storage_submission_8b.json
```

Test 6: MLPerf Storage Submission (70B Model)

Purpose: A heavier version of the MLPerf stress test using a large model to generate a more intense I/O load, further testing the limits of the storage subsystem.

```
python3 kv-cache.py \
    --model llama3.1-70b-instruct \
    --num-users 40 \
    --duration 600 \
    --gpu-mem-gb 0 \
    --cpu-mem-gb 4 \
    --generation-mode realistic \
    --performance-profile throughput \
    --seed 42 \
    --output mlperf_v3_storage_submission_70b.json
```

Test 7: RAG Workload Simulation

Purpose: Simulates a Retrieval-Augmented Generation (RAG) workload, which involves a write-heavy ingestion phase followed by bursty, high-throughput reads. This is an excellent stress test for RAG-specific applications.

```
python3 kv-cache.py \
    --model llama3.1-8b \
    --num-users 30 \
    --duration 300 \
    --gpu-mem-gb 16 \
    --cpu-mem-gb 32 \
    --enable-rag \
    --generation-mode realistic \
    --cache-dir /mnt/nvme \
    --seed 42 \
    --output results_rag_workload.json
```

Test 8: Maximum Stress (The "Kitchen Sink")

Purpose: This is the ultimate stress test. It combines the largest model (70B), the I/O-intensive RAG workload, and the capacity-seeking autoscaler to find the absolute maximum throughput your system can handle when every demanding feature is enabled.

```
python3 kv-cache.py \
    --model llama3.1-70b-instruct \
    --num-users 10 \
    --duration 300 \
    --gpu-mem-gb 16 \
    --cpu-mem-gb 64 \
    --enable-rag \
    --enable-autoscaling \
    --autoscaler-mode capacity \
    --generation-mode realistic \
    --cache-dir /mnt/nvme \
    --seed 42 \
    --output results_max_stress.json
```