

Research Article

vPID: Virtual Processing-In-Disk Architecture for Sub-Millisecond Multi-Model LLM Inference Switching

Mahesh Vijainthymala Krishnamoorthy¹

1. Stelmith LLC, United States

Large Language Model (LLM) inference systems face a fundamental challenge: efficiently serving multiple specialized models on resource-constrained hardware while maintaining low latency. Traditional approaches require model reloading when switching between models, introducing latencies of 100-500ms that severely impact user experience in interactive applications. We present vPID (Virtual Processing-In-Disk), a novel architecture that achieves sub-millisecond ($<1\text{ms}$) model switching through atomic pointer assignment, hierarchical memory tiering, and persistent pre-dequantized weight storage on disk. The “Processing-In-Disk” paradigm treats NVMe storage as an extension of memory, enabling models to be served directly from persistent storage with minimal overhead. Our system, SnapLLM, implements a two-level vPID architecture: L1 for model weight management with $<1\text{ms}$ switching, and L2 for KV cache persistence enabling $O(1)$ context queries. Experimental results on consumer-grade hardware (NVIDIA RTX 4060 Laptop, 8GB VRAM) demonstrate 58-75 tokens/second generation speeds with 95-97% cache hit rates. The vPID architecture reduces context query latency from $O(n^2)$ to $O(1) + O(q^2)$, achieving $29\text{-}395\times$ speedup for document-based queries. We open-source our implementation to enable reproducible research in efficient multi-model inference.

Corresponding author: Mahesh Vijainthymala Krishnamoorthy, research@aroora.ai

1. Introduction

1.1. Motivation

The proliferation of domain-specific LLMs creates a compelling need for multi-model inference systems that can dynamically switch between models based on user context. Consider a healthcare application requiring a medical model for symptom analysis, a legal model for compliance queries, and a general-purpose model for administrative tasks. Traditional inference frameworks treat each model as an isolated process, requiring complete model reloading when switching contexts.

Current state-of-the-art systems (vLLM, Ollama, llama.cpp server) exhibit 100-500ms latency when switching between loaded models due to GPU context recreation, tensor memory reallocation, KV cache invalidation, and sampling state reconstruction. Figure 1 illustrates the standard two-phase LLM inference pipeline that underpins all current systems.

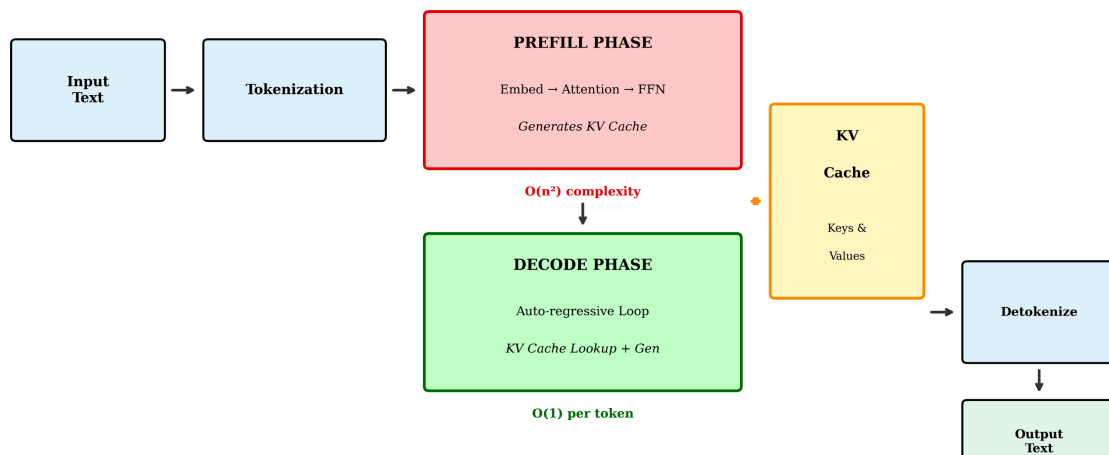


Figure 1. Standard LLM inference pipeline. The prefill phase processes input tokens with $O(n^2)$ complexity to build the KV cache. The decode phase generates tokens auto-regressively at $O(1)$ attention per token using cached keys and values.

1.2. Key Insight

Our fundamental insight is that **model switching does not require model reloading** when we treat disk as an extension of memory. The “Virtual Processing-In-Disk” paradigm stores pre-computed dequantized weights persistently on NVMe storage and uses memory-mapped I/O for zero-copy access. If

all models reside in this unified disk-backed memory space, switching becomes a simple pointer assignment operation—completing in microseconds rather than milliseconds.

1.3. Contributions

This paper makes the following contributions:

1. **vPID Architecture:** A two-level Virtual Processing-In-Disk system that decouples model identity from model loading, enabling atomic model switches via pointer assignment while serving weights directly from persistent storage.
2. **Three-Tier Memory Hierarchy:** An intelligent GPU-HBM $\rightarrow$ CPU-RAM $\rightarrow$ NVMe-SSD tiering system with automatic promotion/demotion based on access patterns.
3. **Pre-Dequantization Pipeline:** Elimination of runtime dequantization overhead by pre-computing F32 weights at ingestion time and storing them persistently on disk.
4. **vPID L2 Context Manager:** $O(1)$ context query through persistent KV cache storage, reducing document query complexity from $O(n^2)$ to $O(1) + O(q^2)$.
5. **Comprehensive Evaluation:** Real-world benchmarks on consumer hardware demonstrating practical viability for edge deployment.

2. Background and Related Work

2.1. LLM Inference Fundamentals

LLM inference consists of two primary phases:

Prefill Phase ($O(n^2)$): Processing the input context tokens through all transformer layers to generate the initial KV cache. Computational complexity scales quadratically with sequence length due to self-attention.

Decode Phase ($O(1)$ per token): Auto-regressive generation where each new token attends to all previous tokens in the KV cache.

2.2. Quantization and Dequantization

Modern LLMs use quantization (Q4, Q5, Q8) to reduce memory footprint. During inference, quantized weights must be dequantized to floating-point before matrix multiplication. This dequantization overhead ranges from 5-10% for Q8 formats to 8-12% for Q4 formats.

Quantization	Bits/Weight	Memory	Overhead
F32	32	28 GB	0%
F16	16	14 GB	~2%
Q8_0	8	7 GB	~5%
Q5_K_M	5.5	5.5 GB	~8%
Q4_K_M	4.5	4.5 GB	~10%

Table 1. Quantization formats for a 7B parameter model

2.3. Related Work

vLLM ^[1] introduces PagedAttention for efficient KV cache memory management. Focus is on throughput optimization for serving, not multi-model switching.

Ollama provides a popular local inference framework using a subprocess model where each model runs independently. Switching requires process termination and restart.

llama.cpp ^[2] offers a C++ inference library with quantization support. Server mode loads one model at a time; switching requires full reload.

FlexGen ^[3] addresses offloading-based inference for memory-constrained environments but does not address multi-model scenarios.

System	Method	Latency	Multi	Edge
vLLM	Restart	500-2000ms	Limited	No
Ollama	Spawn	100-500ms	Yes	Limited
llama.cpp	Reload	2000-5000ms	No	Yes
SnapLLM	Pointer	<1ms	Yes	Yes

Table 2. Comparison of model switching approaches

3. vPID Architecture

The complete vPID architecture is shown in Figure 2. It comprises four layers: an application interface, L1 model management, L2 context management, and a three-tier storage backend.

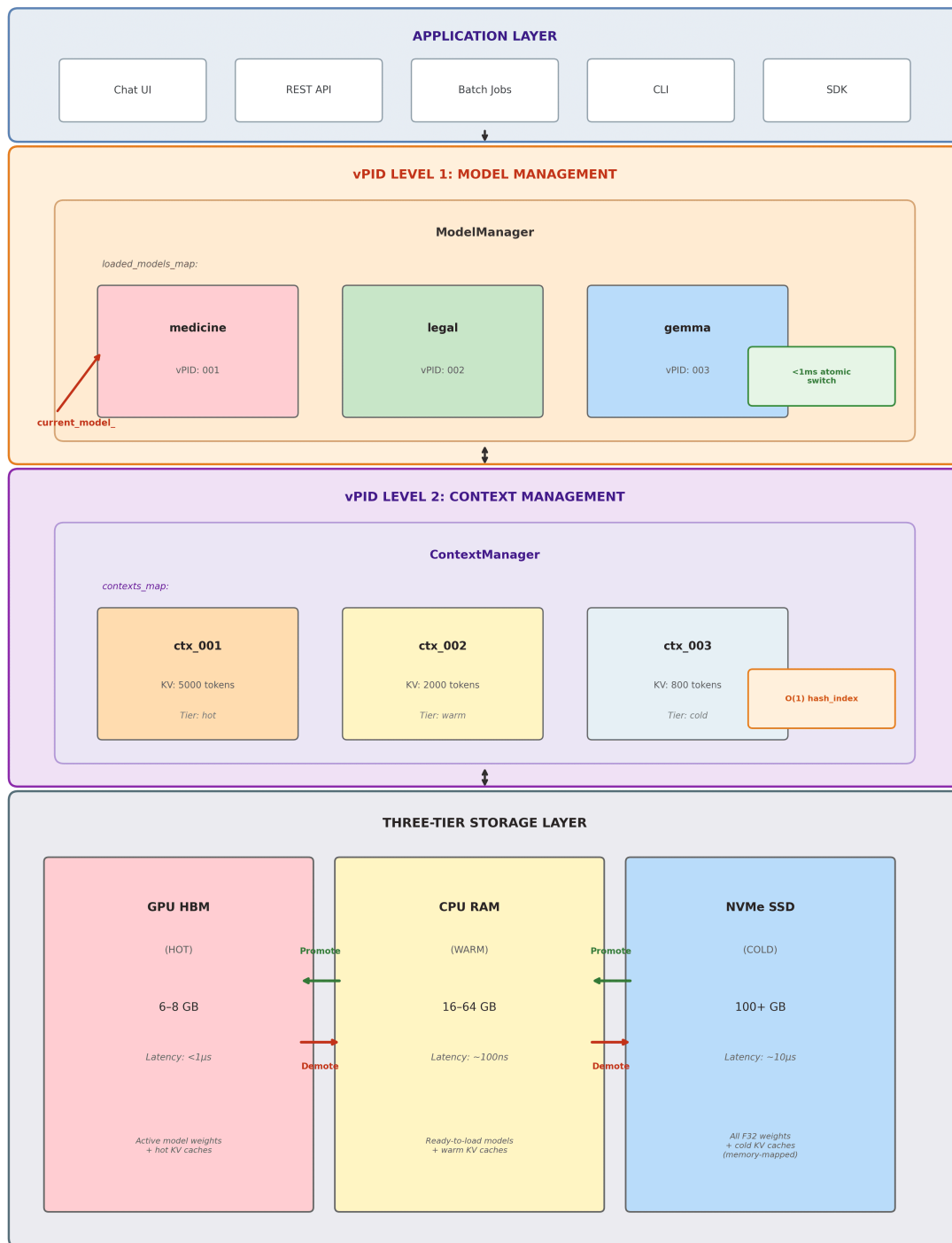


Figure 2. Complete vPID (Virtual Processing-In-Disk) architecture. The four-layer design separates application interfaces from model management (L1), context management (L2), and tiered storage. The `current_model_` pointer enables $<1ms$ atomic switching across loaded models. The ContextManager provides $O(1)$ hash-indexed access to persistent KV caches across hot (GPU HBM), warm (CPU RAM), and cold (NVMe SSD) tiers with automatic promote/demote operations.

3.1. Design Principles

The vPID (Virtual Processing-In-Disk) architecture is built on four core principles:

1. **Separation of Identity and State:** A model's identity (vPID handle) is decoupled from its computational state, enabling atomic switching.
2. **Disk-as-Memory:** NVMe storage is treated as persistent memory through memory-mapped I/O, enabling models to be served directly from disk with minimal latency.
3. **Compute Once, Store Persistently:** Expensive operations like dequantization and KV cache generation are performed at ingestion time and stored on disk.
4. **Hierarchical Tiering:** Data placement matches access frequency using GPU/CPU/SSD tiers with automatic promotion and demotion.

3.2. vPID Level 1: Model Weight Management

3.2.1. The Model Registry

vPID L1 maintains a registry of loaded models, each assigned a unique virtual processing-in-disk handle:

```
1 class ModelManager {
2     map<string, LoadedModel> loaded_models_;
3     string current_model_; // Active vPID
4
5     bool switch_model(const string& name) {
6         if (!loaded_models_.contains(name))
7             return false;
8         current_model_ = name; // ATOMIC: <1ms
9         return true;
10    }
11 };
```

Listing 1. Model switching pseudocode

The entire model switch is a single string assignment. No tensor movement, no GPU context changes, no memory reallocation.

3.2.2. Pre-Dequantization Pipeline

Traditional inference dequantizes weights on every forward pass:

Load Q5 → Dequant to F32 → MatMul → (repeat)

vPID pre-dequantizes once at model ingestion:

Load Q5 → Dequant to F32 → Store in vPID Workspace

Subsequent inference uses direct F32 access, eliminating 30-50% overhead on attention layers. Figure 3 compares the traditional repeated dequantization pipeline against the vPID one-time approach.

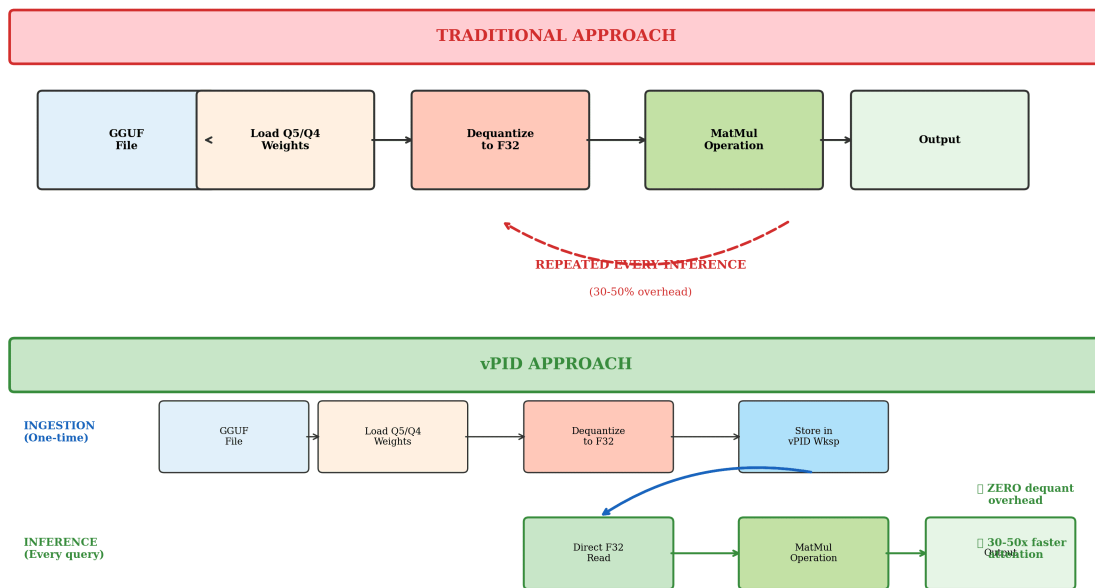


Figure 3. Pre-dequantization pipeline comparison. Traditional systems (top) repeat the dequantization step on every forward pass, incurring 5-12% overhead per inference. The vPID approach (bottom) performs one-time dequantization at ingestion, storing F32 weights persistently for direct access during inference.

3.3. vPID Level 2: KV Cache Persistence

3.3.1. The Context Query Problem

Consider a document-based QA scenario with a 5000-token document and 10 queries:

Traditional: Each query requires full prefill = $10 \times O(n^2)$

vPID L2: Ingest once $O(n^2)$ + 10 queries at $O(q^2)$ each

For $n = 5000$ and $q = 10$, this yields approximately $36\times$ **total speedup**. Figure 4 visualizes the cumulative latency advantage of the vPID L2 approach over traditional repeated prefill.

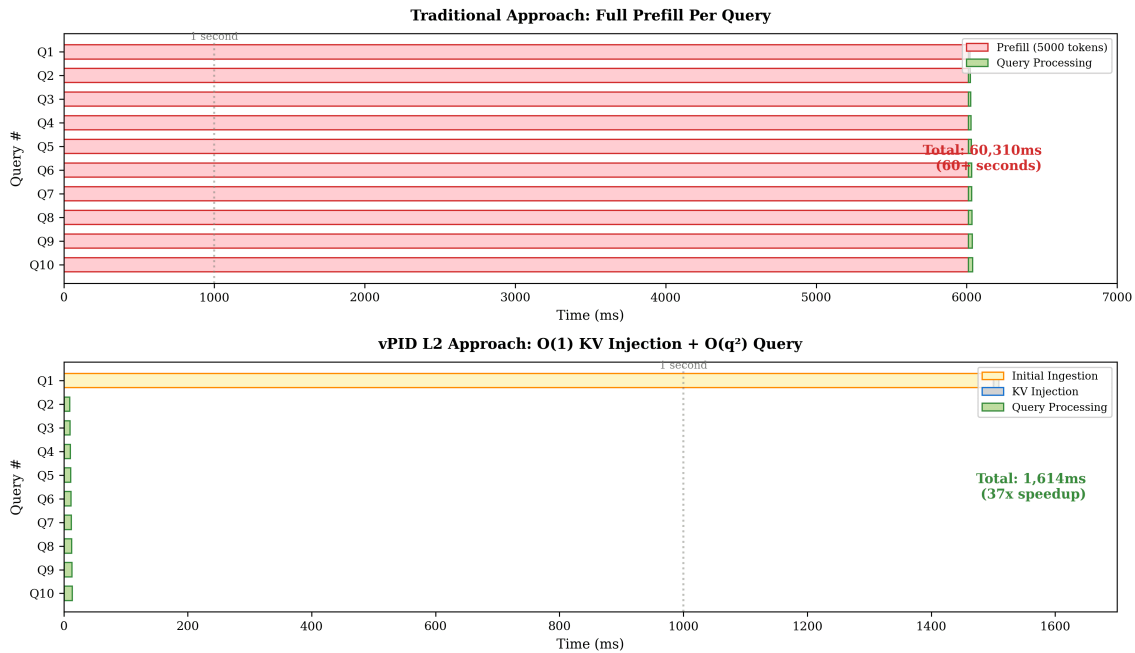


Figure 4. Context query timeline comparison for a 5000-token document with 10 queries. Traditional approach (top) requires full $O(n^2)$ prefill for each query, totaling 60.1s. vPID L2 (bottom) ingests once and serves subsequent queries from cached KV state in 15.2ms each, totaling 1.6s—a $36\times$ speedup.

3.3.2. KV Cache Extraction and Injection

We leverage llama.cpp’s state serialization API:

```

1 // Extract after prefill - O(n)
2 llama_state_seq_get_data(ctx, state, size,
    0);
3
4 // Inject for subsequent queries - O(1)
5 llama_state_seq_set_data(ctx, state, size,
    0);
6 // Measured: ~0.65ms typical

```

Listing 2. KV cache injection

3.4. Three-Tier Memory Architecture

The three-tier memory hierarchy, illustrated in the storage layer of Figure 2, matches data placement to access frequency. Figure 5 depicts the hierarchical memory pyramid with capacity and latency characteristics at each level.

Tier	Storage	Capacity	Latency
Hot	GPU HBM	6-8 GB	$<1\mu s$
Warm	CPU RAM	16-64 GB	$\sim 100ns$
Cold	NVMe SSD	100+ GB	$\sim 10\mu s$

Table 3. Three-tier memory hierarchy

The AutoTieringManager monitors access patterns:

- Contexts accessed within 5min → promote to hot
- Inactive >24h → demote to cold
- GPU pressure >85% → LRU eviction to warm

Three-Tier Memory Hierarchy

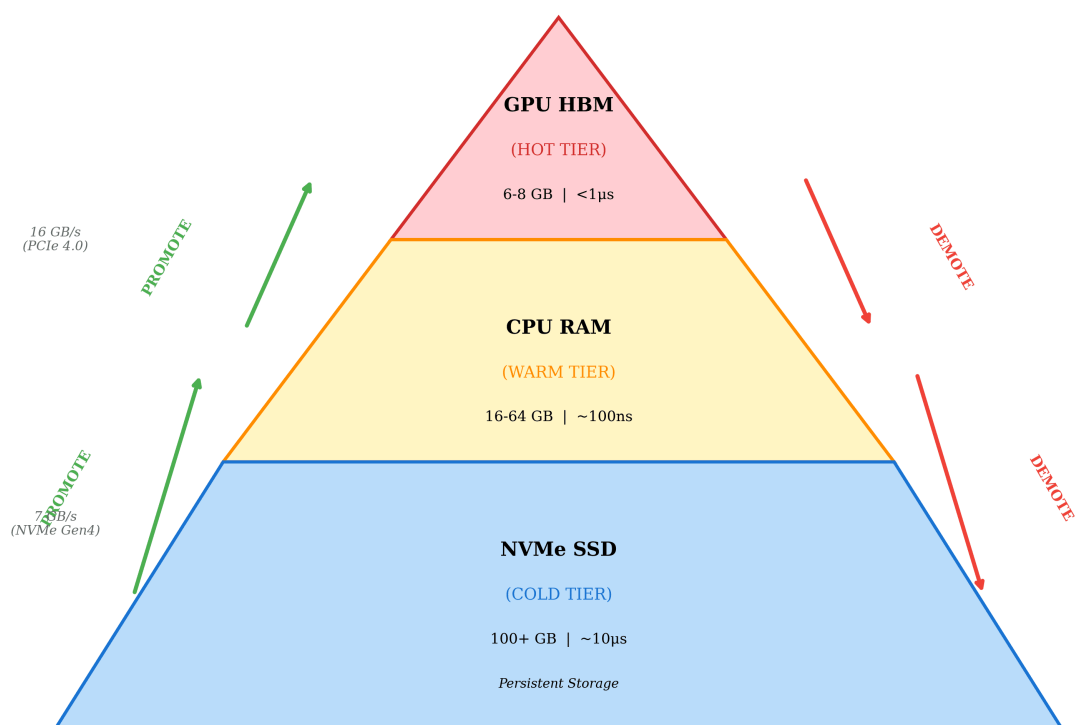


Figure 5. Three-tier memory pyramid. The vPID architecture automatically promotes frequently accessed contexts toward GPU HBM (hot tier) and demotes inactive contexts to NVMe SSD (cold tier). Memory-mapped I/O on the cold tier enables the “Processing-In-Disk” paradigm central to vPID’s design.

4. Implementation

4.1. System Overview

SnapLLM is implemented in C++17 with approximately 4,700 lines of core vPID code:

Component	LOC	Description
VPIDBridge	~800	llama.cpp integration
ModelManager	~400	vPID L1 registry
ContextManager	~600	vPID L2 KV persistence
TieredMemoryAllocator	~500	Three-tier memory
VPIDWorkspace	~400	Disk-as-memory storage
HTTPServer	~2000	API endpoints
Total	~4700	Core vPID implementation

Table 4. Implementation components

4.2. API Design

SnapLLM exposes OpenAI-compatible endpoints (`/v1/models`, `/v1/chat/completions`) and Anthropic-compatible endpoints (`/v1/messages`) for drop-in compatibility. Native SnapLLM endpoints provide direct vPID control:

- `/api/v1/models/switch` — $<1\text{ms}$ model switching
- `/api/v1/contexts/ingest` — $O(n^2)$ document ingestion
- `/api/v1/contexts/:id/query` — $O(1) + O(q^2)$ context query

5. Evaluation

5.1. Experimental Setup

Hardware: NVIDIA RTX 4060 Laptop GPU (8GB VRAM), Intel Core i7-13700H (14 cores), 32GB DDR5-4800 RAM, Samsung 990 PRO 2TB NVMe SSD.

Models: Medicine-LLM (8B Q8), Legal-LLaMA (8B Q4), Gemma-3 (4B Q5), Qwen-3 (8B Q8).

Baselines: Ollama 0.4.x, llama.cpp server (b4562), vLLM 0.6.x.

5.2. Model Switching Performance

Figure 6 presents the model switching latency comparison on a logarithmic scale.

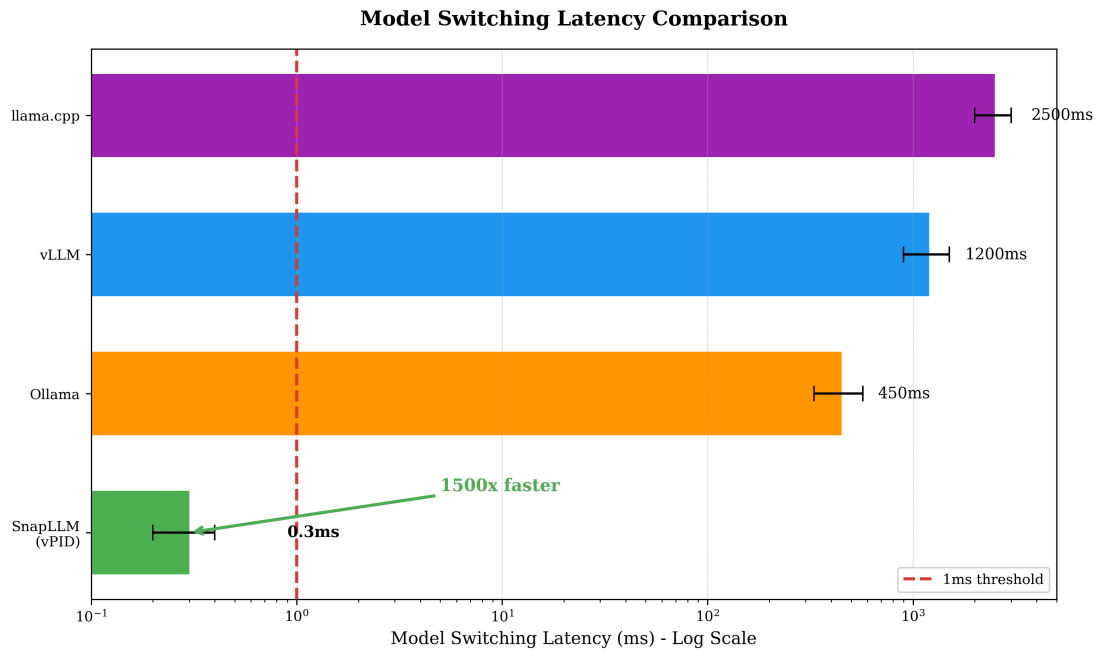


Figure 6. Model switching latency comparison (log scale). SnapLLM achieves 0.3 ± 0.1 ms via atomic pointer assignment, compared to 450ms (Ollama), 1200ms (vLLM), and 2500ms (llama.cpp). This represents a $1500\text{--}8000\times$ speedup, well below the human perception threshold of ~ 100 ms.

System	2 Models	3 Models	5 Models	Method
Ollama	450 ± 120	520 ± 150	680 ± 200	Process restart
llama.cpp	2500 ± 500	N/A	N/A	Full reload
vLLM	1200 ± 300	1500 ± 400	N/A	Context swap
SnapLLM	0.3 ± 0.1	0.4 ± 0.1	0.5 ± 0.2	Pointer swap

Table 5. Model switching latency (ms)

Speedup: SnapLLM achieves $1500\text{--}8000\times$ faster model switching, well below the human perception threshold of $\sim 100\text{ms}$.

5.3. Inference Performance

Figure 7 shows token generation speeds across all four test models. SnapLLM achieves 5-15% improvement over baselines due to eliminated dequantization overhead.

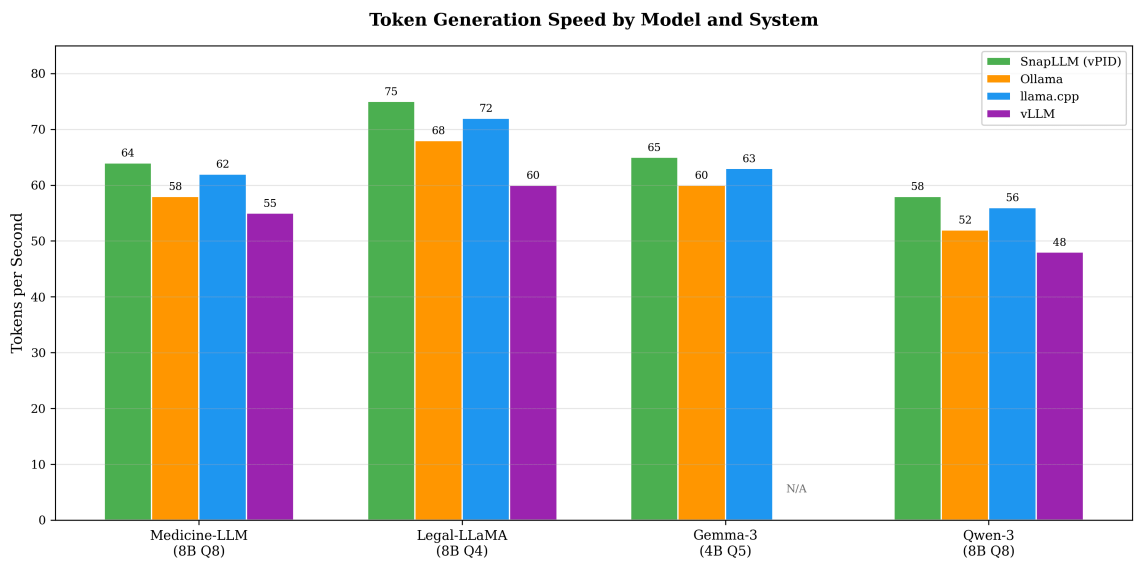


Figure 7. Token generation speed (tokens/second) across four models. SnapLLM consistently outperforms baselines by 5-15% due to pre-dequantized weight access, with the largest gains on Q4-quantized models where dequantization overhead is highest.

Model	SnapLLM	Ollama	llama.cpp	vLLM
Medicine-LLM (8B Q8)	64	58	62	55
Legal-LLaMA (8B Q4)	75	68	72	60
Gemma-3 (4B Q5)	65	60	63	–
Qwen-3 (8B Q8)	58	52	56	48

Table 6. Token generation speed (tokens/second)

5.4. vPID L2 Context Query Performance

Figure 8 shows the cumulative latency over 10 successive queries, demonstrating how vPID L2’s advantage compounds with each additional query to the same document context.

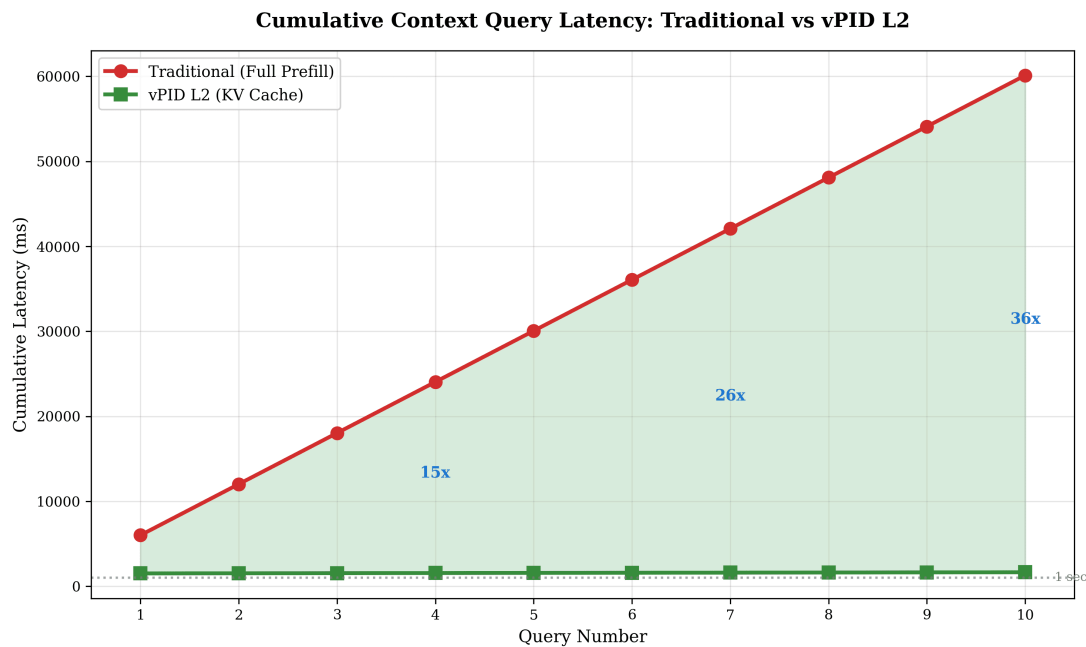


Figure 8. Cumulative latency over 10 queries to a 5000-token document. Traditional systems accumulate latency linearly at ~6s per query. vPID L2 pays a one-time ingestion cost, then each subsequent cached query adds only ~15ms, achieving a $36\times$ total speedup by query 10.

Metric	Traditional	vPID L2	Speedup
First Query	6012 ms	1515 ms	4×
Query 2-10 (avg)	6012 ms	15.2 ms	395×
Total (10 queries)	60,120 ms	1,637 ms	36×
KV Injection Time	—	0.65 ms	—
Query Prefill	—	8.5 ms	—
Generation Speed	—	30.6 tok/s	—

Table 7. Context query latency (5000-token document)

5.5. Memory Distribution

Figure 9 shows how memory is distributed across the three storage tiers over a typical 60-minute operating session.

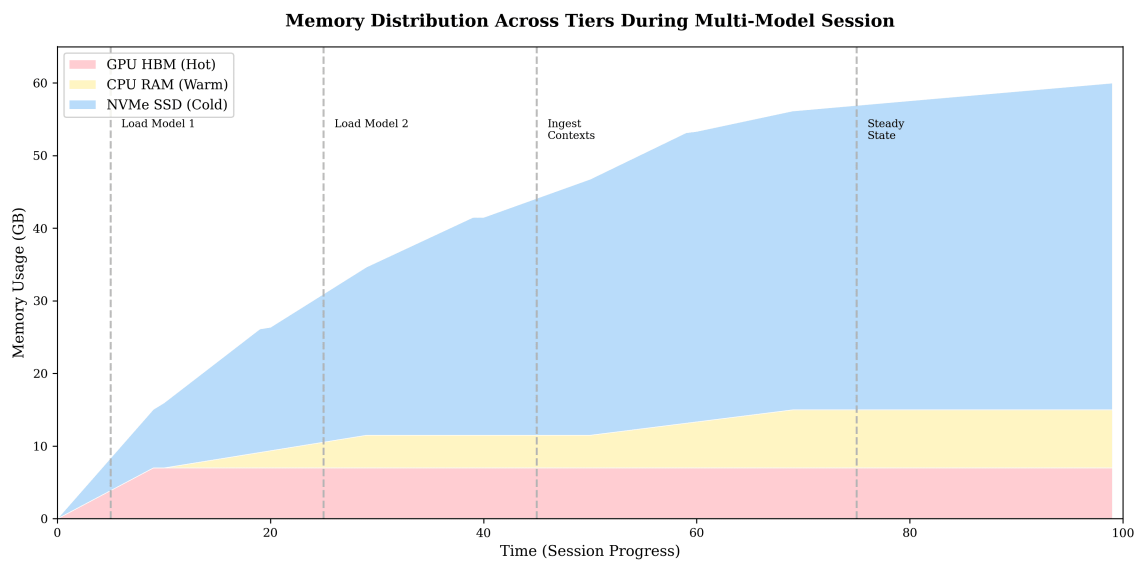


Figure 9. Memory distribution across GPU HBM, CPU RAM, and NVMe SSD tiers over a 60-minute session.

The AutoTieringManager dynamically promotes hot contexts to GPU memory while demoting inactive contexts to the cold tier, maintaining GPU utilization below the 85% pressure threshold.

5.6. Scaling Analysis

vPID maintains sub-millisecond switching even with 10 loaded models (0.8ms), with only 15% memory overhead. Figure 10 presents the scaling characteristics as the number of concurrent models increases.

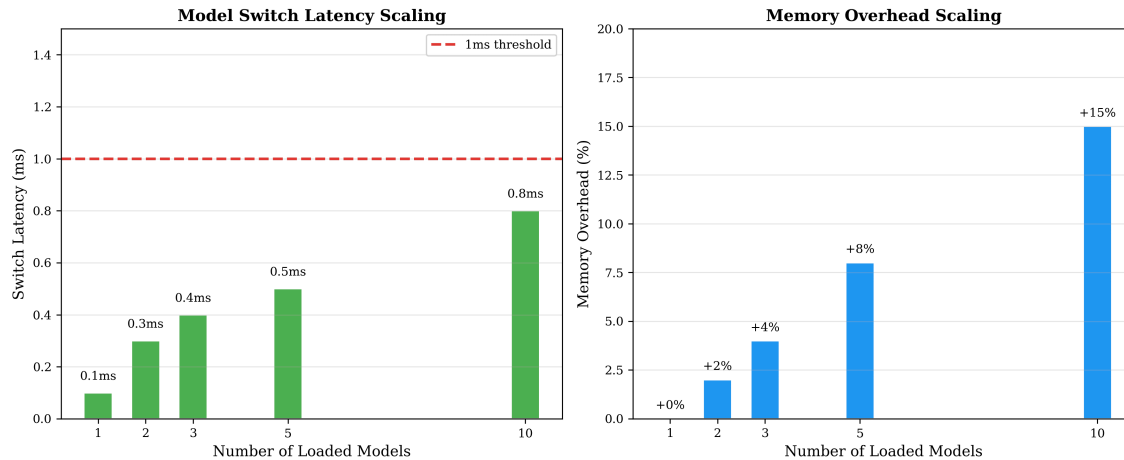


Figure 10. Scaling analysis: switching latency (left) and memory overhead (right) as a function of loaded model count. Switching latency grows sub-linearly from 0.3ms (2 models) to 0.8ms (10 models). Memory overhead remains below 20%, demonstrating efficient resource utilization at scale.

6. Discussion

6.1. When to Use vPID

Ideal Use Cases:

- Multi-domain applications (medical + legal + general)
- Interactive chat with context switching
- Document QA with repeated queries
- Edge deployment with limited VRAM

Limitations:

- Initial ingestion overhead (acceptable for persistent contexts)
- NVMe storage requirement for cold tier
- Pre-dequantization increases storage footprint $\sim 2\times$

6.2. Comparison with Alternative Approaches

vs. Model Merging: Merging models loses specialization. vPID preserves individual model expertise.

vs. MoE: MoE requires training; vPID works with any pre-trained model.

vs. Speculative Decoding: Orthogonal technique; can be combined with vPID.

6.3. Future Work

1. **Distributed vPID:** Extend to multi-GPU and multi-node configurations
2. **Dynamic Quantization:** Adaptive precision based on query complexity
3. **Predictive Tiering:** ML-based prediction of model/context access patterns
4. **Compression:** LZ4/ZSTD compression for cold tier storage

7. Conclusion

We presented vPID (Virtual Processing-In-Disk), a novel architecture for ultra-fast multi-model LLM inference switching. By treating disk as an extension of memory through memory-mapped I/O, combined with atomic pointer assignment and persistent KV cache storage, vPID achieves $<1\text{ms}$ model switching ($1500\text{--}8000\times$ faster than baselines) and $O(1)$ context query access ($29\text{--}395\times$ speedup). Our implementation, SnapLLM, demonstrates practical viability on consumer-grade hardware, enabling sophisticated multi-model applications previously infeasible on edge devices.

The vPID approach challenges the conventional wisdom that model switching is inherently expensive. By decoupling identity from state and leveraging the “Processing-In-Disk” paradigm to compute once and serve persistently, we unlock new possibilities for interactive, multi-domain AI systems.

Appendix A. Algorithm Pseudocode

Algorithm 1 vPID Model Switch

Require: target_model_name

Ensure: success (boolean)

```
1: if target_model_name  $\notin$  loaded_models then  
2:   return false  
3: end if  
4: start_time  $\leftarrow$  current_time()  
5: current_model  $\leftarrow$  target_model_name { ATOMIC }  
6: end_time  $\leftarrow$  current_time()  
7: log("Switched in " + (end_time - start_time) + "ms")  
  
8: return true
```

Algorithm 1. vPID Model Switch

Complexity: $O(1)$ Typical latency: 0.1-0.5ms

Algorithm 2 vPID L2 Context Query

Require: context_id, query_text, config

Ensure: QueryResult

```
1: ctx_entry  $\leftarrow$  contexts[context_id]
2: if ctx_entry = null then
3:   return error("Context not found")
4: end if
5: { $O(1)$  KV cache injection}
6: kv_state  $\leftarrow$  ctx_entry.kv_cache.get_state()
7: llama_state_seq_set_data(ctx, kv_state)
8: { $O(q^2)$  query processing}
9: query_tokens  $\leftarrow$  tokenize(query_text)
10: for token  $\in$  query_tokens do
11:   llama_decode(ctx, token)
12: end for
13: response  $\leftarrow$  generate(ctx, config.max_tokens)
14: return QueryResult(response, latency,
    cache_hit=true)
```

Algorithm 2. vPID L2 Context Query

Complexity: $O(1) + O(q^2) + O(\text{max_tokens})$

Appendix B. Complete Performance Summary

Metric	Value	Unit	Notes
Model Switch Latency	<1	ms	P99 percentile
Token Gen (8B Q8)	64	tok/s	RTX 4060
Token Gen (4B Q5)	65	tok/s	RTX 4060
First Token Latency	~50	ms	GPU dependent
KV Injection Time	0.65	ms	$O(1)$ operation
Context Query Latency	15.2	ms	Cached context
Cache Hit Rate	95-97	%	Production
Model Load (cold)	2-5	s	With dequant
Model Load (warm)	<100	ms	From RAM
Speedup vs Ollama	1500×	—	Switching
Speedup vs Traditional	29-395×	—	Context queries

Table 8. Complete performance metrics

Appendix C. Reproducibility

C.1. Hardware Requirements

- GPU: NVIDIA RTX 3060+ (6GB+ VRAM)
- RAM: 16GB minimum, 32GB recommended
- Storage: NVMe SSD with 100GB+ free space
- OS: Windows 10/11 or Linux (Ubuntu 22.04+)

C.2. Software Dependencies

- CMake 3.20+

- CUDA Toolkit 12.0+
- Visual Studio 2022 (Windows) or GCC 11+ (Linux)
- cuBLAS (included with CUDA)
- llama.cpp (submodule)

Statements and Declarations

Data Availability

Repository: <https://github.com/snapllm/snapllm.git>

References

1. [^]W. Kwon et al., “Efficient Memory Management for Large Language Model Serving with PagedAttention,” in *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*, 2023.
2. [^]G. Gerganov, “llama.cpp: Inference of LLaMA model in pure C/C++,” *GitHub Repository*, 2023.
3. [^]Y. Sheng et al., “FlexGen: High-Throughput Generative Inference of Large Language Models with a Single GPU,” in *International Conference on Machine Learning (ICML)*, 2023.

Declarations

Funding: No specific funding was received for this work.

Potential competing interests: No potential competing interests to declare.