Towers of Segments: Hierarchical KV Cache Storage from Commodity Workstations to Production Servers

Gregory Magarshak
gmagarshak@faculty.ienyc.edu
Safebots, Inc. & IE New York City
New York, USA

ABSTRACT

Today, the KV cache state for a single 4000-token conversation with a 70B-parameter language model occupies ~ 1.28 GB. Multi-tenant LLM services treat this as a structural cost of doing business: storage for conversation memory is genuinely expensive, and "indefinite history" is a premium feature offered grudgingly because the underlying state cannot be cheaply retained. We show that this cost is an artifact of how cache state is represented, not a fundamental property of the workload, and that with the right architecture per-session storage drops to approximately ~ 1.5 KB—a compression of 5×10^5 **to** 1.5×10^6 **over raw FP16 KV**, supported by two pillars: the data processing inequality (the token sequence is a sufficient statistic for KV state given fixed model weights) and arithmetic coding's near-optimal achievement of source entropy.

The compression stack has three coupled stages: *structural amortization* (shared base segments stored once per deployment, with per-session storage converging to the dynamic block at rate $O(1/N)$); *depth-adaptive quantization* (segments at the base of the hierarchy receive aggressive bit reductions justified by an $O(\alpha_\ell^{1/2}\cdot 2^{-b_\ell})$ attention-output bound); and *arithmetic coding against the PLT* (each segment encoded asynchronously as a bitstring against P_M estimated by a small distilled language model, approaching $\sim$ 2–5 bits per token rather than ~ 320 KB per token). The empirical foundation is established: Chinchilla 70B achieves 0.664 bits per byte on enwik9 via arithmetic coding (~ 2.7 bits/token), Llama2-13B yields 75% bit savings over gzip, and small distilled models perform comparably on natural language. We introduce *Towers of Segments*, the storage architecture that realizes all three stages: cached state organized as a tree of segments composed at request time via RoPE-shifted assembly, with arithmetic-coded compressed segments thawed on demand by running one parallel prefill pass and reinserted into the cache trie for unlimited subsequent reuse and cross-session sharing.

The practical picture is striking. For a 70B model at 4000-token sessions, the realistic operating point is $\sim$ 850,000× per-session compression—from ~ 1.28 GB down to ~ 1.5 KB—bounded by the perplexity of the source rather than the size of the KV state. A million compressed sessions occupy ~ 1.5 GB of storage; an entire year's conversation history for a 100,000-user platform fits on a consumer thumb drive. A 64 GB MacBook with a 4 TB SSD holds ~ 2.7 billion compressed sessions, exceeding any realistic deployment, with a ~ 2.7 GB decoder bundle (a distilled $\tilde{M}$ plus the model's embedding table) pinned in unified memory. The cold-tier thaw cost is one parallel prefill pass per session reactivation ($\sim$ 100–500 ms on commodity hardware)—paid once when a long-idle session resumes, after which the segment is back in the hot cache for the rest of the session, shared via the cache trie across any other sessions whose towers reference it. As an ambitious theoretical extension, we conjecture (grounded in LAWS-style certified experts [3], sampling the PLT on actual production workflows) the existence of a universal cache decoder $\mathcal{U}_M$ that maps tokens directly to the full all-layer KV state in time bounded by a single transformer layer's compute, reducing thaw latency by an additional order of magnitude.

The economic consequence is significant: cache storage stops being a binding constraint on LLM service design. Indefinite conversation memory, full transcript retention, user-portable history, archival of every session a user has ever had with a chatbot—all become cheaper than the text content of the conversations themselves. The cache wall, treated rigorously as an information-theoretic problem, is well within the perplexity-bit envelope of the source distribution, and the architecture to reach it fits on a personal laptop.

KEYWORDS

KV cache, transformer inference, RoPE, multi-tenant serving, hierarchical storage, shared memory, probabilistic language tries

1 INTRODUCTION

Consider a developer running an 8B-parameter language model on a 32 GB laptop, wanting to serve a small chatbot deployment—perhaps a few hundred concurrent users distributed across a dozen bots and a handful of communities. Each session at 4,000-token context occupies approximately 500 MB of KV state at FP16; sixty active sessions saturate the available RAM. With model weights and working buffers accounted for, the practical limit is closer to forty. The developer concludes that serving more than a few dozen users requires a server with more RAM, a GPU, or both.

This conclusion is wrong, and it is wrong by orders of magnitude. The same workstation can hold several thousand warm sessions with the architecture we present in this paper. The path is structural: in the deployment we just described, hundreds of sessions share the same community prompt, dozens share the same bot prompt, every session shares the platform base. If shared content is stored physically once and amortized across all sessions that use it, per-session

storage no longer scales linearly with the size of the deployment. It approaches the per-session *dynamic content*—the transcript—and everything above the transcript pays for itself in proportion to the number of tenants using it.

The architecture we propose, *Towers of Segments*, realizes this scaling concretely. Cached state is organized as a tree of segments. Each session's full context (its "tower") is materialized at request time by composing pre-cached segments. The composition uses rotary position embedding's algebraic structure to relocate segments to their tower-specific positions without re-running prefill. Combined with depth-adaptive quantization (more aggressive at deeper bases where attention concentrates less) and the per-segment compression of prior work [2, 5], per-session storage drops to approximately $|B_{\text{dyn}}|$ asymptotically.

What the workstation case looks like. On a 32 GB machine with an 8B model at Q4 weights, after the full compression stack, per-session storage settles at roughly 6 MB at moderate N. The 32 GB budget—less the 4 GB of model weights and a few GB for working buffers—holds approximately 3,000 warm sessions. CPU-only inference benefits the most: progressive base-segment quantization is more valuable when RAM is the binding constraint than when HBM is, and the cache trie with RoPE-shifted segment composition reuses computation across many requests without any GPU acceleration. Pure CPU inference at 5–15 tokens/sec on a modern x86 workstation is enough to drive several concurrent low-traffic conversations from the warm slot pool, materializing the right tower for each request in microseconds.

What scales up. The same architecture deploys to a multi-TB-RAM production server with a 70B model and serves a hundred thousand warm sessions. The asymptotic scaling is the same; only the constants differ. Between the workstation and the production server, there is a continuum of intermediate deployments—a 256 GB developer server holds several thousand 70B-model sessions; a 1 TB multi-socket server holds tens of thousands. The architecture spans three orders of magnitude of hardware scale with the same theoretical foundation.

Why this hasn't been done. Existing compression methods reduce per-session cost but not its scaling with deployment size. TurboQuant [5] achieves $\approx 5\times$ per-vector compression and proves a per-vector Shannon lower bound. Predictive delta coding [2] compresses the sequential structure of a single session's cache, bounded by per-token surprisal. Exact prefix sharing [6, 7] deduplicates byte-identical prefixes, valuable for sessions sharing a system prompt but limited to byte-exact matches and unable to exploit hierarchy depth. None of these change the per-session $\Theta(L)$ scaling.

The opportunity has been hiding because the analysis was always done at the session level. At the deployment level, hierarchical structure provides a strictly stronger compression resource: the per-session cost approaches $|B_{\text{dyn}}|$ rather than $\Theta(L)$, and the convergence rate is $O(1/N)$.

Contributions.

(1) A formal storage model (Section 3) defining towers, segments, the cache trie, and RoPE-shifted composition.
(2) Materialization Correctness Theorem (Section 4): RoPE-shifted compositional towers are exact in real arithmetic under a relative-position invariance condition that holds whenever segments are constructed in a strictly hierarchical pipeline.
(3) Asymptotic Per-Tenant Storage Theorem (Section 5): per-session storage cost approaches $|B_{\text{dyn}}|$ as $N \to \infty$, with explicit constant-factor scaling under a branching-process model.
(4) Depth-Adaptive Quantization Theorem (Section 6): quantization error in segment σ_ℓ contributes $O(\alpha_\ell^{1/2} \cdot 2^{-b_\ell})$ to the attention output, justifying aggressive quantization at deeper bases.
(5) Hierarchy Composition Theorem (Section 7): the four-layer compression stack exploits pairwise disjoint sources of redundancy and composes multiplicatively.
(6) Implementation paths (Section 8) on stock llama.cpp via slot file caching, with shared-memory swap for GPU deployments and dedicated CPU-only deployment patterns for workstations.
(7) Arithmetic coding against the PLT (Section 10): segments encoded as bitstrings against P_M, approaching the entropy bound of ~ 4 bits per token, with parallel Layer-0 decompression in single-layer compute time. Cross-tenant clustering (Section 11) and universal model-shipped decoders (Section 12) extend the same machinery to PLT-proximate sharing and per-model decoder artifacts.

The trajectory. The paper builds in stages. Sections 3–7 present the basic Towers architecture with structural amortization, depth-adaptive quantization, and orthogonal composition with prior compression layers—reducing per-session storage from $\Theta(L)$ to approximately $|B_{\text{dyn}}|$. Section 8 grounds this in concrete deployment patterns from 32 GB workstations to multi-TB servers. Sections 10–12 then push compression to the source-coding lower bound via arithmetic coding against the model's own distribution, with Layer-0 parallel decompression as the cache primitive that makes this practical. The combined stack delivers approximately 850,000× per-session compression over raw FP16 KV (1.28 GB to 1.5 KB on a 70B model at 4000-token sessions), bringing storage cost within the perplexity-bit envelope of the source. A 64 GB MacBook with a 4 TB SSD holds approximately 2.7×10^9 compressed sessions, more than any realistic deployment will use.

Relation to prior work. The PLT-based compression program of [1–3] provides the theoretical scaffolding for prefix structure, sequential entropy bounds, and certified substitution. This paper adds the storage-layout axis, exploiting the multi-tenant amortization that single-session compression cannot reach. The Materialization Correctness Theorem requires byte-stable segment construction; pipelines satisfying

this include the write-time context assembly of [4] and simpler deterministic template-based construction, but Towers is independent of any specific construction architecture.

2 BACKGROUND

2.1 Probabilistic Language Tries

The PLT framework [1] defines a probabilistic language trie $T(M)$ over vocabulary V as the rooted directed tree whose nodes are prefixes and whose edges from $x \in V^*$ are labeled by tokens $t \in V$ with weight $P_M(t \mid x)$. The trie metric $d_T(s, s') = -\log_2 P_M(s \wedge s')$ measures distance in probability space. The PLT framework's role here is twofold: in the basic architecture, conceptual (the cache trie introduced in Section 3 draws on the same intuition that prefix structure is the resource, but is operationally distinct); and in the extension sections (Sections 10–12), $T(M)$ itself becomes a structural compression resource derived directly from model weights.

2.2 Modular Attention Reuse

The most directly related line of work is modular attention state reuse, beginning with PromptCache [11]. PromptCache introduced the foundational technique: schema-defined prompt modules whose KV states are precomputed once and reused across requests, with discrete position-ID assignment and RoPE lookup-table adaptations to handle modules appearing at different positions in different prompts. PromptCache reports 1.5–10× TTFT latency reduction on GPU and 20–70× on CPU, and observes that CPU inference benefits more from the technique than GPU inference.

A substantial follow-up literature has extended this approach. CacheBlend [12] enables non-prefix KV cache reuse by selectively recomputing a small subset of cross-attention tokens to restore accuracy after positional reassignment. TurboRAG [13] introduces RoPE-consistent position reordering for chunk stitching in RAG pipelines. KVLink [14] adds positional re-encoding for KV cache reuse across stored documents. ChunkAttention [15] structures the prefix cache as an auxiliary prefix tree, similar to our cache trie. SGLang's RadixAttention [7] uses a radix tree for cross-request prefix caching. LMCache [16] provides an enterprise-scale KV cache layer integrating with vLLM and SGLang for hierarchical storage. TokenDance [17] introduces block-sparse diff storage for multi-agent KV cache sharing, demonstrating 11–17× compression on collective-sharing workloads. KVShare [18] and SemShareKV [19] address multi-tenant reuse with selective recomputation for accuracy preservation.

This paper's position relative to that lineage. The core technique—precomputing KV states for reusable prompt segments and composing them at inference time with positional adjustment—is established prior art. We do not claim a novel mechanism. Our contributions are theoretical and architectural, layered on top of the existing technique:

(1) The *Asymptotic Per-Tenant Storage Theorem* (Section 5) is a formal scaling result for multi-tenant deployments not present in prior modular-attention-reuse work. PromptCache and follow-ups demonstrate per-request speedup; we analyze how per-session storage cost converges to $|B_{\text{dyn}}|$ as deployment size grows.

(2) The *Depth-Adaptive Quantization Theorem* (Section 6) gives a specific bound on attention error from quantization at hierarchy depth, justifying the progressive quantization scheme (Q4/Q6/Q8). Prior work treats quantization of cached states as future work.

(3) The *Hierarchy Composition Theorem* (Section 7) establishes orthogonal multiplicative composition of four compression layers (per-vector quantization, predictive delta coding, prefix deduplication, tower structural sharing).

(4) Deployment patterns: SSD-paged warm pools with Lindy-style cold-to-hot promotion (Section 8.3); unified-memory tower assembly on Apple Silicon (Section 8.3); CPU-only deployments using SIMD-friendly progressive base quantization.

(5) Integration with the PLT-based compression program: sections 10, 11, and 12 develop deeper compression by leveraging the PLT derived from model weights.

The RoPE shifting we describe in Section 4 is algebraically equivalent to the position-ID-based approach of PromptCache: both achieve the same end (relocating cached KV state to new logical positions) via mathematically equivalent operations. We use the rotation-algebra formulation because it admits a clean proof of materialization correctness under the relative-position invariance condition.

2.3 Streaming Attention and Long-Context

A parallel line of work [9, 10] manipulates RoPE position encoding to extend effective context length and preserve attention sinks. The position-shift operation we use shares technical structure with these methods, but the application differs: we use it for cross-session segment composition, not within-session context extension.

2.4 Sequential KV Compression

KV Sequential [2] establishes the bound $H(\text{KV}_i \mid \text{KV}_{<i}) \leq H(t_i \mid t_{<i})$: the conditional entropy of the i-th KV vector is bounded by the per-token surprisal. Two compression layers exploit this: probabilistic prefix deduplication (segment-level) and predictive delta coding (within-segment). We use the surprisal bound in Section 6 to motivate depth-adaptive quantization, and extend the framework in Section 10 to bound compression of segments via PLT-coordinate representation.

2.5 Byte-Stability and Segment Construction

Towers' segments are byte-stable artifacts: the segment representing a shared platform base must produce identical bytes whenever it is constructed for any session using it. Without this property, segments cached for one session cannot be

reused for another. Multiple pipelines satisfy byte-stability. The write-time context assembly of [4] satisfies it by construction; simpler deterministic template-based pipelines satisfy it trivially. We treat byte-stability as an assumption on the upstream pipeline rather than a dependency on any specific architecture.

3 FORMAL MODEL

3.1 Segments and Towers

Definition 1 (Segment)**.** A *segment* $\sigma = (\tau_\sigma, \pi_\sigma, \mathrm{KV}_\sigma)$ consists of: a token sequence $\tau_\sigma \in V^*$; a source position $\pi_\sigma \in \mathbb{N}$ at which τ_σ was processed; and the KV state KV_σ produced by that processing, layered and head-decomposed in the model's standard format.

Definition 2 (Tower)**.** A *tower* $\Theta = (\sigma_1, \ldots, \sigma_d)$ is an ordered chain of segments whose token sequences concatenate to form a session's full token context. Segment σ_i occupies logical positions $[p_i, p_i + |\tau_{\sigma_i}|)$ where $p_i = \sum_{j<i} |\tau_{\sigma_j}|$.

Definition 3 (Cache Trie)**.** The *cache trie* $\mathcal{T}_C$ is the rooted directed tree whose nodes are segments and whose edges encode the extension relation: $\sigma \to \sigma'$ if the segment-construction pipeline admits σ' as a continuation of σ. Each path from root to leaf corresponds to one tower. Segments at depth ℓ from the root are *level-ℓ segments*.

Remark 4 (Cache trie vs. PLT)**.** The cache trie $\mathcal{T}_C$ is not the PLT $T(M)$. PLT nodes are prefixes weighted by model probability; cache trie nodes are physical segments with positions, storage, and admissibility relations determined by the segment-construction pipeline. Cache trie boundaries are architectural choices, not properties of M.

3.2 Storage and Sharing

Definition 5 (Storage cost)**.** The total storage cost of serving a population of towers $\{\Theta^{(n)}\}_{n=1}^{N}$ is

$$C_{\text{storage}} = \sum_{\sigma \in \bigcup_n \Theta^{(n)}} |\mathrm{KV}_\sigma|,$$

counting distinct segments once. A segment shared across k towers contributes once, not k times.

3.3 RoPE Position Shifting

We assume the underlying model uses rotary position embedding [8]: at every layer ℓ, $k_p^{(\ell)} = R(p) \cdot \tilde{k}_p^{(\ell)}$ where $\tilde{k}_p^{(\ell)}$ is the position-stripped K vector and $R(p)$ is a block-diagonal rotation matrix.

Definition 6 (RoPE shift)**.** The *RoPE shift* from position p to p' is the operation $\mathrm{RoPE}_{p \to p'} : k \mapsto R(p' - p) \cdot k$. It is linear, isometric (rotations preserve norms), and exact in real arithmetic.

Remark 7 (Relative-position invariance)**.** The attention score satisfies $\langle q_p, k_{p'} \rangle = \tilde{q}_p^\top \cdot R(p - p') \cdot \tilde{k}_{p'}$, depending only on the relative position $p - p'$. This is the structural property that makes segment composition possible: a segment shifted by a constant Δ retains correct attention behavior provided its preceding context shifts by the same Δ.

Remark 8 (V vectors do not rotate)**.** RoPE applies only to keys. Cached V vectors copy without modification when relocating a segment.

4 MATERIALIZATION CORRECTNESS

Theorem 9 (Materialization Correctness)**.** *Let $\Theta = (\sigma_1, \ldots, \sigma_d)$ be a tower with source positions $(\pi_{\sigma_1}, \ldots, \pi_{\sigma_d})$ and target positions $(p_1, \ldots, p_d)$ where $p_i = \sum_{j<i} |\tau_{\sigma_j}|$. Suppose the* relative-position invariance condition *holds: for each segment σ_i, the tokens preceding σ_i at relative offsets $(-1, -2, \ldots, -p_i)$ in the target tower are identical to those at the same relative offsets when σ_i was computed. Let $\mathrm{KV}^{compose}$ be the cache assembled by RoPE-shifting each segment's K vectors from π_{σ_i} to p_i and copying V vectors. Let $\mathrm{KV}^{prefill}$ be the cache from monolithic prefill on the concatenated tokens. Then $\mathrm{KV}^{compose} = \mathrm{KV}^{prefill}$ in real arithmetic; in FP16/BF16 the per-element error is bounded by $d \cdot \varepsilon_{rot} \cdot \|\mathrm{KV}\|_2$.*

Proof. *Step 1.* In RoPE attention, the layer-ℓ hidden state at position p is determined by (i) the token sequence $(t_1, \ldots, t_p)$ and (ii) the relative positional structure. The second is invariant under uniform position shifts; the first depends only on the token content. By induction on layers, hidden states are functions of tokens and relative positions only.

Step 2. The position-stripped K vector $\tilde{k}_p^{(\ell)}$ is determined by the hidden state. Under a uniform shift by Δ, both the tokens and the relative positions are preserved, so $\tilde{k}_p^{(\ell)} = \tilde{k}_{p+\Delta}^{(\ell)}$. The full K vector at the shifted position is $k_{p+\Delta}^{(\ell)} = R(\Delta) \cdot k_p^{(\ell)}$, exactly $\mathrm{RoPE}_{p \to p+\Delta}(k_p^{(\ell)})$. V vectors depend only on the hidden state and are identical under shift.

Step 3. The premise ensures each segment's preceding-token context matches between source and target at every relative offset. By Step 1, hidden states inside the segment are identical between contexts. By Step 2, K vectors transform correctly and V vectors are unchanged.

Step 4. Boundary continuity: at the boundary between σ_i and σ_{i+1}, the attention at the first position of σ_{i+1} attends to the last positions of σ_i. The premise guarantees these tokens are identical between source and target; Step 2 guarantees their KV vectors transform correctly.

For the FP16 bound, per-rotation error is $\varepsilon_{\text{rot}} = O(\epsilon_{\text{fp}} \cdot \|k\|)$. Errors do not amplify across d segments (rotations are isometric) but accumulate at most linearly by triangle inequality. □

Remark 10 (When the premise holds)**.** The premise—identical relative-position-indexed tokens preceding each segment—is automatic when segments are constructed in a strictly hierarchical pipeline: σ_ℓ is always preceded by the same parent chain. The cache trie's structure encodes this constraint. Any pipeline producing byte-stable hierarchical segments satisfies the premise.

Corollary 11 (FP16 noise tolerance)**.** *For typical FP16 rotation error $\varepsilon_{rot} \approx 2^{-10} \cdot \|k\|$ and tower depth $d \leq 8$, the accumulated error is $\leq 8 \cdot 2^{-10} \cdot \|\mathrm{KV}\|_2 \approx 2^{-7} \cdot \|\mathrm{KV}\|_2$, well within FP16 quantization noise. Multi-segment composition is numerically safe in standard inference precisions.*

Remark 12 (Position-disjoint, not layer-disjoint)**.** Segments are disjoint in position range within a tower, not in model layers. Every layer produces K and V vectors for every position; a segment stores the full per-layer KV state for its position range.

5 ASYMPTOTIC PER-TENANT STORAGE

This section establishes the paper's central deployment result: per-session storage cost approaches the dynamic-block cost alone as N grows, with shared base segments amortizing to zero.

Proposition 13 (Aggregate Storage)**.** *For a population of N towers over a cache trie $\mathcal{T}_C$ with S_ℓ distinct segments at depth ℓ and per-segment storage cost ℓ_ℓ, total storage is*

$$C_{storage}^{total} = \sum_{\ell=0}^{d-1} S_\ell \cdot \ell_\ell.$$

Proof. Each distinct segment contributes once by the storage definition. Summing over depths yields the bound. □

Theorem 14 (Asymptotic Per-Tenant Storage)**.** *Let the cache trie be generated by a branching process with mean branching factor β_ℓ at depth ℓ, and let $|B_{\mathrm{dyn}}|$ be the per-session dynamic content size. Assume $N \gg \prod_{\ell=0}^{d-2} \beta_\ell$. Then the per-session storage cost satisfies*

$$\frac{C_{storage}^{total}}{N} \;=\; |B_{\mathrm{dyn}}| \;+\; \sum_{\ell=0}^{d-2} \frac{\ell_\ell \cdot \prod_{j<\ell} \beta_j}{N},$$

which converges to $|B_{\mathrm{dyn}}|$ as $N \to \infty$. The convergence rate is $O(1/N)$.

Proof. Under the branching process, $\mathbb{E}[S_\ell] = \prod_{j<\ell} \beta_j$ for $\ell < d-1$ (the non-leaf levels), independent of N. At the leaf level (transcript), every session has its own distinct segment: $S_{d-1} = N$. By Proposition 13, total storage is

$$C_{\mathrm{storage}}^{\mathrm{total}} = \sum_{\ell=0}^{d-2} \ell_\ell \cdot \prod_{j<\ell} \beta_j \;+\; N \cdot |B_{\mathrm{dyn}}|.$$

Dividing by N gives the per-session cost. The non-leaf terms scale as $\prod_j \beta_j / N \to 0$ as $N \to \infty$, while the leaf term is constant in N. □

Corollary 15 (Convergence to dynamic-only)**.** *For any fixed branching pattern, per-session storage converges to $|B_{\mathrm{dyn}}|$ at rate $O(1/N)$. The constant in this $O(1/N)$ is $\sum_\ell \ell_\ell \prod_{j<\ell} \beta_j$—the total non-leaf storage—which depends on the hierarchy's branching structure but is independent of N.*

Table 1: Per-session storage (MB) at varying deployment size N, for a 70B model with GQA ($H_{kv} = 8$, $d_{\mathbf{head}} = 128$), tower of depth 4 with 1000-token segments at FP16 ($\approx$ 1 GB per segment), branching $(\beta_0, \beta_1, \beta_2) = (1, 50, 10)$, and $|B_{\mathrm{dyn}}| = 1$ GB. Convergence to the $|B_{\mathrm{dyn}}| = 1024$ MB floor is monotone in N.

N	B_{perm}/sess	B_{sess}/sess	B_{bot}/sess	total/sess
500	2.0	102	1024	2152
5,000	0.2	10.2	102	1137
50,000	0.02	1.02	10.2	1035
500,000	0.002	0.10	1.02	1025

Table 2: Deployment scenarios at varying hardware. "Warm" sessions live in RAM; "cold" sessions live on SSD/NVMe and page in on demand. The MacBook row exploits unified memory (no PCIe transfer for GPU access) and the 4 TB internal SSD as warm-pool extension. The architecture is feasible across four orders of magnitude of hardware scale.

Hardware	RAM	Model	Sess. cost	Sess. capacity
Workstation	32 GB	8B Q4	6 MB	∼3,000 warm
MacBook + 4 TB SSD	64 GB	70B Q4	40 MB	∼100,000
Dev server	128 GB	8B Q4	6 MB	∼20,000 warm
GPU server	256 GB	70B Q4	40 MB	∼5,000 warm
Multi-socket	1 TB	70B Q4	40 MB	∼25,000 warm
High-density	4 TB	70B Q4	40 MB	∼100,000 warm

Remark 16 (Reading the scaling)**.** Table 1 makes the asymptotic claim concrete. At $N = 500$ sessions, the platform-base contribution per session is $\approx$ 2 MB; the community-base contribution is $\approx$ 100 MB; the bot-base contribution is $\approx$ 1 GB; the dynamic content is $\approx$ 1 GB; total $\approx$ 2.1 GB. At $N = 500{,}000$, all three base contributions have dropped to a few MB combined, and the per-session cost is essentially $|B_{\mathrm{dyn}}|$. *The architecture's benefit scales with deployment size, not against it.*

Remark 17 (Realistic numbers with full compression stack)**.** Applying depth-adaptive quantization (Section 6) and predictive delta coding [2] to the dynamic block reduces each constant by quantization factors. For Q4/Q6/Q8 base quantization and a ∼30× KV Sequential compression on the dynamic block, per-session storage at $N = 10^5$ becomes:

- B_{perm} at Q4: $\approx$ 5 KB/session
- B_{sess} at Q6: $\approx$ 380 KB/session
- B_{bot} at Q8: $\approx$ 5 MB/session
- B_{dyn} compressed: $\approx$ 35 MB/session

Total per session: $\approx$ 40 MB, versus $\approx$ 4 GB at FP16 without sharing. A 100× reduction in absolute terms, with the structural sharing factor growing at large N and the compression factors constant.

Remark 18 (The architecture works at every scale)**.** Table 2 makes the operational claim concrete. A 32 GB workstation—commodity hardware—serves a few thousand warm sessions on an 8B model with the full compression stack. The architecture is not exclusively a large-deployment optimization; it is the right way to organize cache storage at any scale where there is hierarchical structure in the prompt set. Larger RAM budgets serve proportionally more sessions, but the architectural advantages (per-session cost approaching $|B_{\text{dyn}}|$, structural sharing scaling with N) are present even at the small end.

6 DEPTH-ADAPTIVE QUANTIZATION

The progressive quantization of base segments is correctness-preserving: deeper segments receive smaller queries' aggregate attention, so quantization error contributes less to the attention output. We make this precise.

Theorem 19 (Depth-Adaptive Quantization)**.** *Consider a tower with segment σ_ℓ containing n_ℓ tokens, whose K vectors are quantized to b_ℓ bits per component, introducing per-component error $\|\delta k_j\|_2 \le \varepsilon_\ell = O(2^{-b_\ell} \cdot \|k\|_2)$. For a query q_i at any later position, with aggregate attention weight $\alpha_\ell^{(i)} = \sum_{j \in \sigma_\ell} w_j$ onto σ_ℓ, the first-order contribution to attention output from σ_ℓ's quantization errors is bounded by*

$$\|\delta a_i^{(\sigma_\ell)}\|_2 \;\le\; \frac{\varepsilon_\ell \cdot \|q_i\|_2}{\sqrt{d_{head}}} \cdot \sqrt{\alpha_\ell^{(i)}} \cdot \sqrt{M_\ell},$$

where $M_\ell = \sum_{j \in \sigma_\ell} w_j \|v_j - a_i\|_2^2$ is the weighted second moment of value residuals on σ_ℓ.

Proof. Scaled dot-product attention is $a_i = \sum_j w_j v_j$ with $w_j = \exp(q_i \cdot k_j / \sqrt{d_{\text{head}}})/Z$. Perturbing $k_j \to k_j + \delta k_j$ for $j \in \sigma_\ell$:

$$\delta a_i = \sum_{j \in \sigma_\ell} \frac{w_j (q_i \cdot \delta k_j)}{\sqrt{d_{\text{head}}}} (v_j - a_i)$$

where the $(v_j - a_i)$ factor arises from softmax normalization. Applying Cauchy-Schwarz to $q_i \cdot \delta k_j \le \|q_i\| \varepsilon_\ell$ and then to the weighted sum:

$$\sum_j w_j \|v_j - a_i\|_2 \le \sqrt{\sum_j w_j} \cdot \sqrt{\sum_j w_j \|v_j - a_i\|^2} = \sqrt{\alpha_\ell^{(i)}} \cdot \sqrt{M_\ell}$$

yields the stated bound. □

Remark 20 (Why deep bases tolerate aggressive quantization)**.** For deep base segments, queries from the transcript attend weakly: $\alpha_\ell \ll 1$. The error scales as $\alpha_\ell^{1/2}$, so the same bit-budget reduction at deeper segments produces less attention drift than at recent segments. Concretely, halving the bits at a segment where $\alpha_\ell = 0.01$ produces $10\times$ less error than halving the bits at a segment where $\alpha_\ell = 1$.

Corollary 21 (Three-tier discrete scheme)**.** *A practical allocation:*

- *Platform base (B_{perm}, level 0): Q4 KV*
- *Community base (B_{sess}, level 1): Q6 KV*
- *Bot base and transcript: Q8 or FP16*

yields ≈ 2–3× reduction over uniform Q8 with quality cost bounded by Theorem 19 at each level.

Remark 22 (Connection to PLT surprisal)**.** The PLT framework [1] characterizes a node's depth by $-\log_2 P_M(s)$; high-probability prefixes have lower information content. KV Sequential [2] bounds per-position KV entropy by surprisal. Depth-adaptive quantization aligns with both: it spends fewer bits where the underlying information content is provably lower.

7 HIERARCHY COMPOSITION

Theorem 23 (Hierarchy Composition)**.** *The four compression layers*

- *(Q) per-vector quantization [5],*
- *(D) predictive delta coding [2],*
- *(P) prefix deduplication [2],*
- *(T) tower structural sharing (this paper),*

exploit pairwise disjoint sources of redundancy and compose multiplicatively:

$$C_{storage}^{full} \le \frac{C_{storage}^{raw}}{\rho_Q \cdot \rho_D \cdot \rho_P \cdot \rho_T}.$$

Proof. The four layers operate on distinct axes. (Q) compresses each stored vector treated as isolated. (D) compresses the residual of each vector against the model's prediction. (P) decides which segments need to be stored at all. (T) determines how stored segments are laid out across many tenants.

$Q \perp D$: (D) reduces magnitude; (Q) quantizes whatever is stored. The residuals are still vectors subject to per-vector quantization. Established in [2, Prop. 3].

$Q, D \perp P, T$: (Q) and (D) operate on the content of each stored segment. (P) and (T) determine which segments exist and where they live. Distinct axes.

$P \perp T$: (P) and (T) are conjugate: (P) is the indicator function determining $|\bigcup_n \Theta^{(n)}|$ (set membership); (T) is the multi-tower layout efficiency (how the membership set is realized in storage). The savings from (P) are encoded in the segment count; the savings from (T) are the layout factor for that count.

Each layer operates on the output of the others without exploiting the same redundancy twice; the four factors compose. □

Remark 24 (Realistic stacking, asymptotic)**.** For a 70B model at $N = 10^5$ sessions with hierarchical structure $(1, 50, 10)$:

- Raw FP16 per session: ≈ 4 GB
- Structural sharing (T) at large N: per-session base cost → 0, total ≈ 1 GB per session
- + Per-vector quantization (Q) at 5×: ≈ 200 MB
- + Predictive delta coding (D) at 30×: ≈ 35 MB
- + Depth-adaptive quantization on bases: shaves another ≈ 5 MB

The total stack delivers $\approx 100\times$ per-session reduction, with the structural-sharing component continuing to improve as N grows.

8 IMPLEMENTATION

We describe two implementation paths. The first works on stock llama.cpp without code modifications and achieves the structural sharing benefit. The second adds shared-memory swap for GPU servers, accessing slot storage at PCIe bandwidth via host-mapped pages.

8.1 Stock llama.cpp via Slot File Caching

The llama-server binary supports persistent slot save/restore via `--slot-save-path`, exposing `POST /slots/<id>/save` and `POST /slots/<id>/restore` endpoints. Slot files are binary serializations of the full KV state for one slot.

The recipe:

(1) **Pre-cache the platform base.** Prefill on B_{perm} tokens with B_{perm} as the prompt; save the resulting slot file as `Bperm.bin`. Costs one prefill ($\approx$ 1 s for 4000 tokens on H100).
(2) **Pre-cache each community base.** For each community C: restore `Bperm.bin`, run prefill on $B_{\text{sess},C}$ tokens (which extends the cached KV state with $B_{\text{sess},C}$'s content), save as `Bperm_Bsess_C.bin`. Costs one prefill per community.
(3) **Pre-cache each bot base.** For each bot B in community C: restore `Bperm_Bsess_C.bin`, run prefill on $B_{\text{bot},B}$ tokens, save as `full_bot_B.bin`. The result contains $B_{\text{perm}} + B_{\text{sess}} + B_{\text{bot}}$.
(4) **Serve sessions.** A new session for bot B restores `full_bot_B.bin`, then processes the session's transcript turn-by-turn as standard continuations.

What this gives without code changes. The compute benefit of structural sharing: a new session pays prefill only on the dynamic content, not on the full $B_{\text{perm}} + B_{\text{sess}} + B_{\text{bot}}$ stack. Cold-start latency drops from seconds (4000-token prefill) to tens of milliseconds (slot restore).

What this does not give without code changes. The naive recipe stores each `full_bot_B.bin` independently, so the on-disk footprint is the same as per-session caching. To realize the storage savings, slot files must be extracted into deltas—a CPU-side utility ($\approx$ 500 lines of C++) reads the byte layout of the slot file (KV state laid out per layer, per head, per position) and saves only the bytes corresponding to the segment's incremental positions. At session start, the utility splices parent slot bytes with the delta bytes to reconstruct the full slot, which is then restored.

The delta-extraction approach yields the storage scaling of Theorem 14. Risk: the slot file format is not a public API and may change between llama.cpp versions; production deployments must track upstream changes.

8.2 CPU-Only Deployment on Commodity Hardware

The architecture is genuinely accessible without a GPU. On a workstation with 32–128 GB of RAM and a modern x86 CPU, the cache trie with RoPE-shifted segment composition supports a multi-tenant deployment that would otherwise be impossible at this hardware tier.

Why CPU-only inference benefits the most. Two reasons.

First, RAM is more precious on a workstation than HBM is on a GPU server. A 32 GB budget leaves little room for slack, so the per-session $40\times$–$100\times$ reduction from the full compression stack moves the deployment from infeasible (a few dozen sessions) to clearly feasible (a few thousand). The same factor on a 256 GB GPU server moves the deployment from feasible to more feasible—a less dramatic operational shift.

Second, depth-adaptive quantization (Theorem 19) interacts well with CPU SIMD paths. The base segments at Q4 are smaller, so they fit in CPU L2 cache more easily during the attention computation. The shorter dot products execute faster per token at lower precision. Quantization is purely a storage win on GPU; on CPU it is also a throughput win.

Pointer-based segment composition. On a single machine without distributed storage, the tower for a session can be assembled directly in process memory rather than through filesystem-mediated slot files. The cache trie holds pointers (or, more precisely, offsets into a memory-mapped segment store) to each segment's KV data. At request time, the runner walks the cache trie for the session's tower, applies RoPE shifts to each segment's K vectors as it copies them into the inference context's KV buffer, and proceeds with attention computation. The composition is a few microseconds per token rather than the milliseconds-per-MB of file-based assembly.

Workflow on a 32 GB workstation. A practical recipe:

(1) **Materialize the segment library.** Run prefill on each distinct segment in its parent context, save the resulting KV state to a memory-mapped file (mmap'd from a Q4/Q6/Q8-quantized representation per Section 6). For a deployment of one platform base, 10 communities, and 50 bots, total materialization time is approximately $60 \times t_{\text{prefill}}$, where t_{prefill} is the prefill cost for one segment.
(2) **Build the cache trie in memory.** An in-process data structure maps tower identifiers to segment-pointer chains. Each chain entry is $(\text{segment\_offset}, \text{source\_position}, \text{target\_position})$.
(3) **Serve sessions.** On a request, look up the session's tower in the cache trie. Assemble the KV context by walking the segment chain, RoPE-shifting K vectors as needed, copying V vectors directly. Run the inference loop. On completion, persist the per-session dynamic state to disk; the base segments remain in the shared memory-mapped store.

(4) **Manage the warm pool.** The segment store is fixed-size (sized to the available RAM budget minus model weights and working buffers). Per-session dynamic state is paged in and out as sessions become active or idle, with LRU eviction. Cold session resumption pays one disk read of the per-session state; the base segments are always warm.

Throughput. CPU-only inference on a modern x86 workstation runs at approximately 5–15 tokens per second for an 8B model at Q4 weights. This is too slow to drive a high-throughput chat service but easily fast enough to maintain several concurrent conversational sessions (each turn of an LLM-mediated conversation lasts seconds to tens of seconds; the user expects latency at the seconds scale). For a workstation serving 3,000 warm sessions, only a small fraction are active simultaneously; the rest are idle and consume RAM but no compute.

8.3 Unified Memory with SSD Paging: The MacBook Case

Apple Silicon (M1/M2/M3/M4 in Pro/Max/Ultra variants) has a property that changes the latency analysis: the GPU and CPU share the same physical RAM. Tower materialization on a MacBook is qualitatively different from materialization on a discrete-GPU server, and the difference is favorable.

Why unified memory matters for Towers. On a traditional GPU server, the slot store lives in host RAM and reaches the GPU via PCIe DMA at ≈ 30 GB/s. Even with shared-memory swap and pinned pages, the PCIe transfer is the bottleneck. On Apple Silicon, the GPU reads the same memory the CPU writes to, with no transfer. The cache trie walk happens on the CPU (assembling the tower's KV state into a buffer), and the GPU accesses that same buffer for attention computation. Bandwidth on M4 Max is ≈ 400–500 GB/s; a 500 MB tower assembly takes ≈ 1 ms rather than ≈ 17 ms on the PCIe path. For latency-sensitive interactive serving, this is the difference between imperceptible and noticeable.

The 4 TB SSD as warm-pool extension. A typical MacBook configuration: 64 GB unified memory, 4 TB internal SSD with ≈ 6 GB/s sequential read. The SSD becomes an effective L2 cache for cold sessions. A cold session's dynamic state—about 35 MB after the full compression stack—pages in from SSD to unified memory in ≈ 6 ms. Combined with the ≈ 1 ms tower assembly time, a fully cold session resumes in well under 10 ms.

Deployment profile. A 64 GB MacBook with the full basic-stack architecture (Sections 5–7):

- Model weights (70B Q4): ≈ 40 GB
- Working buffers and OS overhead: ≈ 8 GB
- Available for hot KV pool: ≈ 16 GB
- Hot sessions resident: ≈ 400 at ≈ 40 MB per session
- Cold sessions on SSD: up to $\approx 100{,}000$ (limited by SSD capacity at ≈ 40 MB compressed dynamic state each)

The total accessible-session pool is determined by SSD capacity, not RAM, because page-in latency is small enough that idle sessions need not be RAM-resident. A single MacBook holds 100,000 sessions of warm state; the architecture trades 6 ms of page-in latency on cold reactivation for two orders of magnitude more concurrent sessions than RAM alone would support.

With Layer 0 arithmetic compression (Sections 10–12). The cold-storage budget changes dramatically. Each cold session occupies ~ 1.5 KB of compressed bitstring rather than ~ 40 MB. The 4 TB SSD now holds approximately 2.7×10^9 sessions, exceeding the population of any plausible deployment. The decoder bundle ($\sim$ 3–4 GB: distilled $\tilde{M}$ plus universal cache decoder $\mathcal{U}_M$ if available, otherwise $\sim$ 2.7 GB for $\tilde{M}$ plus embedding table) occupies unified memory permanently. The available pool for hot compressed sessions is approximately 12–13 GB after model weights (40 GB Q4) plus decoder bundle plus OS overhead (8 GB).

Latency profile: hot raw KV (active sessions): ~ 1 μs. Warm quantized KV (recent inactive): $\sim$ 0.1–1 ms. Cold-tier reactivation: with $\mathcal{U}_M$ available, $\sim$ 15–70 ms (parallel arithmetic decode plus $\mathcal{U}_M$ forward pass producing all-layer KV). Without $\mathcal{U}_M$ (conservative fallback), $\sim$ 100–500 ms (full model prefill on the decoded tokens). Both paths give the same storage compression; only the cold-tier reactivation speed differs.

The MacBook becomes a fully production-quality platform for community-scale or even small-enterprise-scale multi-tenant deployment. Limited not by storage capacity but by inference throughput on active sessions, with cold-tier reactivation cheap enough ($\sim$ 15–500 ms depending on path) that any user-perceptible delay on resuming an idle conversation is below the threshold of interactive frustration.

What this enables. Local agents and personal-assistant deployments where one machine serves many users or many concurrent tasks. A developer running a Safebots-style platform on a personal MacBook can host an entire community of users, with the cache trie's hierarchical structure ensuring that the platform base, community base, and per-bot bases each live exactly once in storage. CPU-side context assembly with Apple Silicon's vector units handles the RoPE shifts efficiently; GPU-side attention computation runs at standard inference speeds, accessing the assembled context through unified memory without transfer overhead.

Tier allocation: recency, branching, and the Lindy heuristic. The system must choose which sessions to keep resident in RAM (hot pool) and which to evict to SSD (cold pool). Three policy ingredients combine well in practice:

- *Recency.* Standard LRU eviction within the hot pool. Recently-touched sessions stay resident.
- *Branching position.* Segments near the root of the cache trie (the platform base, popular community

bases) are shared across many sessions and should be pinned in hot RAM independently of any individual session's recency. The cache trie itself provides the access-pattern data.

- *Lindy-style longevity for cold-to-hot promotion.* For sessions in the cold pool, the question is which to promote back to hot when capacity becomes available. Sessions that have accumulated a long history (many turns over a long wall-clock period) tend to persist; sessions that were created recently and have only a few turns are more likely to be abandoned. Lindy reasoning—using accumulated lifetime as a predictor of remaining lifetime under heavy-tailed lifetime distributions—is the right intuition for promotion priority among similarly-recent sessions. This is an operational heuristic, not a theorem of this paper, but the heavy-tailed lifetime structure of conversational sessions has been observed in production deployments and matches the assumption.

A practical policy: pin platform and community bases in RAM by default; manage the per-session state with LRU within an age-weighted priority structure, where promotion priority from cold to hot is weighted by accumulated session age.

8.4 Shared-Memory Swap for Discrete-GPU Servers

For traditional x86 servers with discrete GPUs (NVIDIA H100, A100, AMD MI300), slot file I/O can be the latency bottleneck. NVMe bandwidth ($\approx$ 7 GB/s for PCIe 4.0 SSDs) is much lower than the $\approx$ 30 GB/s achievable via PCIe DMA from host RAM to GPU. Shared-memory swap takes advantage:

(1) **Mount tmpfs at the slot-save-path.** Linux's `/dev/shm` or a dedicated tmpfs mount backs the slot directory with RAM rather than disk. The llama-server save/restore code is unchanged; it sees a normal filesystem, but reads and writes hit host RAM.

(2) **Pin host RAM for GPU access.** A small llama.cpp patch wraps the slot buffer allocation in `cudaMallocHost` (or the ROCm equivalent), producing pinned memory the GPU can DMA from at full PCIe bandwidth. Without this patch, slot restore still goes through RAM but pays a CPU-side memcpy.

(3) **Tier between RAM and disk.** For deployments where the full segment set exceeds RAM capacity, keep hot segments in tmpfs and cold segments on NVMe. The cache trie is the natural source of tier hints: high-traffic platform and community bases stay in tmpfs; rarely-used bots stay on disk.

Performance implications. A 500 MB slot restore from NVMe takes $\approx$ 70 ms. From tmpfs without pinned memory, $\approx$ 25 ms. From tmpfs with pinned memory and PCIe DMA, $\approx$ 17 ms. (For comparison, the same operation in Apple Silicon unified memory is $\approx$ 1 ms; see Section 8.3.) For a 70B model with 10,000 sessions/second, the difference between the NVMe and pinned-tmpfs paths is the difference between feasible and infeasible.

Capacity planning. Per-session storage from Table 2: at $N = 10^5$ and full compression stack, $\approx$ 40 MB per session. A deployment of 100,000 active sessions uses $\approx$ 4 TB of RAM for warm storage; cold segments overflow to NVMe. A multi-socket server with 6 TB RAM can hold the entire active state in tmpfs.

8.5 vLLM as an Alternative Backend

vLLM's PagedAttention exposes prefix caching at 16-token block granularity. Tower materialization via prefix-cache seeding: run short inferences to populate the prefix cache with each segment's tokens; subsequent requests benefit from automatic byte-identical block matching. Without kernel modification, vLLM does not perform RoPE shifts at block load, so only exact-position-matched segments are reused. This works for the canonical case where each tower's segments are cached at their natural positions; cross-position reuse requires a kernel extension.

9 DISCUSSION

9.1 When the Architecture Pays Off

The asymptotic-per-tenant scaling of Theorem 14 requires three conditions on the workload:

Hierarchical structure. The deployment must have shared prompt content at multiple levels of hierarchy. Workloads where every session has a unique prompt at every level see no benefit.

Sufficient scale. The per-session cost approaches $|B_{\text{dyn}}|$ at rate $O(1/N)$. At $N \approx 100$, structural sharing reduces per-session storage by perhaps 30%; at $N \approx$ 10,000, by 90%+; at $N \approx$ 1,000,000, the per-session cost is essentially $|B_{\text{dyn}}|$. The architecture earns its complexity at scale.

Byte-stable construction. Materialization correctness requires segments to be byte-identical across reuse contexts. Any deterministic pipeline (template-based, write-time-assembled per [4], or hashed-content-based) satisfies this. Pipelines with hidden non-determinism (timestamps in prompts, random-IDs in templates) violate the premise.

9.2 Limitations

RoPE assumption. The materialization theorem assumes rotary position embedding. Most modern open models (Llama, Qwen, Mistral, Gemma, Phi) use RoPE; older absolute-position models (early GPT, T5) do not.

Engineering cost is real. The asymptotic scaling requires the delta-extraction utility or its equivalent; without it, the architecture provides compute savings but not the asymptotic storage benefit. The delta utility is bounded-complexity ($\approx$ 500 lines of C++) but couples to llama.cpp's slot file format, which is not a stable API.

First-order quantization analysis. Theorem 19 bounds the first-order error. Higher-order terms could matter at very low

bit depths ($b < 3$); empirical validation across model families is future work.

Cold-start prefill cost. Each distinct segment must be materialized once. For a hierarchy with $\sum_\ell S_\ell$ segments, that is $\sum_\ell S_\ell$ prefill runs. At realistic scale ($\approx$ 5500 segments for a typical deployment), warmup runs to $\approx$ 90 minutes. Amortized across millions of subsequent sessions.

9.3 Empirical Validation Plan

The four theorems suggest testable measurements:

(1) *Materialization correctness* (Theorem 9): compare composed-tower KV state to monolithic-prefill KV state on a deployed model. Expected: exact match in real arithmetic; FP16-bounded error otherwise.
(2) *Asymptotic scaling* (Theorem 14): on production traffic, measure per-session storage at varying N. Expected: $1/N$ convergence to $|B_{\text{dyn}}|$.
(3) *Depth-adaptive quantization* (Theorem 19): measure attention output drift as a function of segment depth and bit count. Expected: scaling as $\alpha_\ell^{1/2} \cdot 2^{-b_\ell}$.
(4) *Composition* (Theorem 23): measure cumulative storage at each layer of the stack on a real workload.

9.4 Open Problems

Optimal segment boundary placement; online adaptation of bit budgets based on observed attention statistics; combination with LAWS-style certified expert substitution [3], where heavily-shared segments are replaced by experts; extension to non-RoPE position encodings (sliding-window, ALiBi).

10 DEEP COMPRESSION I: ARITHMETIC CODING AGAINST THE PLT

The compression analyzed in Sections 5–7 treats KV state as the unit of storage. Towers, KV Sequential's delta coding, and TurboQuant's per-vector quantization all operate on stored KV vectors directly. The PLT framework [1] suggests a sharper compression target: encode each segment as a bitstring whose length is bounded by the segment's surprisal under P_M, using arithmetic coding against the PLT as the probability model. The KV state is then reconstructed on demand by decoding the bitstring back to tokens, running a single embedding lookup, and entering the model at layer 1.

10.1 The Compress/Decompress Pair

Definition 25 (PLT-arithmetic encoder)**.** The *PLT-arithmetic encoder* E_M maps a token sequence τ to a bitstring $E_M(\tau)$ via arithmetic coding using the conditional distribution $P_M(t_i \mid t_{<i})$ as the probability model at each position. The expected encoded length is

$$\mathbb{E}[|E_M(\tau)|] = \sum_{i=1}^{|\tau|} -\log_2 P_M(t_i \mid t_{<i}) + O(1),$$

matching the entropy of τ under P_M to within a constant.

Definition 26 (PLT-arithmetic decoder)**.** The *PLT-arithmetic decoder* D_M maps a bitstring b back to its token sequence by sequential arithmetic decoding against P_M. Together, $D_M \circ E_M = \text{Id}$ on token sequences in the support of P_M.

Proposition 27 (Compression bound)**.** *For a segment σ with token sequence τ_σ of length n_σ, the encoded representation $E_M(\tau_\sigma)$ has expected length*

$$\mathbb{E}[|E_M(\tau_\sigma)|] \le n_\sigma \cdot \log_2 \text{PP}(M) + O(1),$$

where $\text{PP}(M)$ is the model's perplexity. For typical natural-language content, $\log_2 \text{PP}(M) \in [3, 5]$ bits per token, compared to ≥ 64 bits per token for raw KV at FP16 (after GQA reduction).

Proof. The expected codeword length of optimal arithmetic coding against the model P_M equals the cross-entropy $H(\tau_\sigma \| P_M)$. When τ_σ is drawn from P_M itself (as is the case for content the model considers plausible), this equals $H(\tau_\sigma) \le n_\sigma \log_2 \text{PP}(M)$ by definition of perplexity. The $O(1)$ overhead accounts for arithmetic coding's finite-precision implementation. □

10.2 Why This Goes Below the Per-Vector Floor

The Shannon lower bound on per-vector KV quantization [5] is derived assuming each KV vector is a random sample from some distribution. The bound is tight for that setting. But the KV vector at position i is not a random sample—it is a deterministic function of the model and the preceding tokens. By the data processing inequality,

$$H(\text{KV}_i \mid \text{KV}_{<i}) \le H(t_i \mid t_{<i}, \text{model weights}) = H(t_i \mid t_{<i}),$$

since the model weights are known to both encoder and decoder. Storing the tokens (compressed against the model's own distribution) is therefore tighter than any per-vector compression of the KV state, by the gap between per-vector entropy and per-token surprisal.

For typical natural-language content, this gap is roughly an order of magnitude: KV vectors at FP16 carry ~ 64 bits per token of representational redundancy that is fully determined by the token sequence plus model weights.

10.3 The Lossy Regime: Below Shannon Entropy

The bound in Proposition 27 is the *lossless* compression floor for the token sequence. Two refinements push below it:

Quantization tolerance. The architecture only requires that reconstructed KV state preserves attention output within the bound of Theorem 19. We do not need bit-exact reconstruction. If the encoder rounds rare-token probabilities below a threshold $p_{\min}$ to a single "out-of-vocabulary" codeword, the average codeword shrinks while introducing a bounded reconstruction error. The bound: error rate $\le p_{\min}$, attention-output drift bounded by Theorem 19 scaled by the rate.

Unbounded overflow for rare sequences. Standard fixed-length codes truncate at a maximum codeword size, costing accuracy on outlier sequences. Arithmetic coding has no such limit: the codeword length adapts to the actual surprisal. A 0.001-probability rare event consumes 10 bits; a 0.000001-probability rarer event consumes 20 bits. Both compress correctly. The average codeword length is the source entropy regardless of the tail.

Remark 28 (Why this matters for cache compression)**.** Inference workloads have long-tailed token distributions: most tokens are highly predictable ("the," "a," common function words and connectives), occasionally a domain-specific term appears ("CRISPR," "etcd," a user's name). Arithmetic coding adapts to this perfectly: predictable tokens consume fractional bits; rare tokens consume more. A fixed-bit-per-token scheme either wastes bits on predictable content or truncates on rare content. The PLT-arithmetic encoder gets both right.

10.4 Asynchronous Compression

The encoder E_M requires running an autoregressive forward pass to compute the conditional probabilities at each position. This is comparable in cost to a single prefill of the segment. The architectural insight: *compression runs asynchronously, off the inference critical path.* When a segment is first cached, it has not yet been requested again. The compressor runs in the background between the segment's first use and any subsequent reuse. The compressed representation lands in storage by the time the second request arrives. Encoding cost is amortized over the segment's lifetime usage; for a base segment reused thousands of times, the per-use amortization is negligible.

10.5 Decompression: Thaw and Reuse

The cache-read primitive is straightforward and uses no learned components beyond the small distilled $\tilde{M}$ already needed for arithmetic decoding. When a cold compressed segment is requested:

(1) *SSD/RAM read of the bitstring.* For a $\sim$ 1.5 KB compressed segment, $\sim$ 0.5–1 ms on SSD, negligible from RAM.
(2) *Parallel arithmetic decode.* The bitstring contains a known token sequence; decoding can be parallelized by chunked decompression (encoder commits decoder state at chunk boundaries, so chunks decode independently on separate GPU warps or CPU SIMD lanes). For a 4000-token segment: $\sim$ 5–15 ms.
(3) *Single parallel prefill pass.* Run the model's standard parallel prefill on the decoded tokens, producing the full all-layer KV state. For a 4000-token segment on a 70B model: $\sim$ 100–500 ms on commodity hardware.

This thaw operation is paid *once* per session reactivation. After the thaw, the materialized KV state goes back into the cache trie as a regular hot segment, where it is reused without further cost for the entire duration of the session—and shared across any other sessions whose towers reference the same segment via the structural amortization of Theorem 14. The thaw cost amortizes over both the temporal lifetime of the session and the cross-session reuse pattern of the cache trie.

Remark 29 (Why parallel arithmetic decoding works)**.** Although classical streaming arithmetic decoding is sequential, the compressed bitstring already contains all the information needed to reconstruct the token sequence—we do not need to ask the model "what comes next" autoregressively. Decoding can be parallelized by chunked decompression: divide the bitstring into independent chunks (with chunk boundaries inserted by the encoder), decode each chunk in parallel, then concatenate. Per-chunk overhead is small; the parallelism scales nearly linearly with chunk count.

10.6 The Decoder as Model Artifact

The decoder bundle has two model-level components, shared across all cache reads in a deployment:

(1) The probability approximation $\tilde{M}$. A small auxiliary language model (a distillation of M, typically $\sim$ 1–2% of M's parameters) provides $P_M(t_i \mid t_{<i})$ for arithmetic decoding. Empirical anchor: Chinchilla 70B achieves 0.664 bits per byte on enwik9 via arithmetic coding, and smaller models (SmolLM2-135M, Llama2-13B) achieve 0.63–0.76 bpb on natural English text. For our purposes, a distilled $\tilde{M}$ of 1–2B parameters at Q4 quantization ($\sim$ 500 MB) achieves $\sim$ 3 bits/token average compression.

(2) The chunking and synchronization protocol. The encoder inserts periodic synchronization points (sequence position offsets at which the arithmetic decoder's state is committed), enabling parallel decompression. Chunk size is a tunable parameter trading encoder simplicity for decoder parallelism.

Total decoder bundle: $\sim$ 500 **MB** for the distilled $\tilde{M}$ at Q4 plus the chunking metadata. The model's own embedding table is reused from the existing model weights (no duplication needed). $\tilde{M}$ is loaded once per deployment and resident in unified memory or GPU HBM for the duration.

Remark 30 (An ambitious extension: the universal cache decoder $\mathcal{U}_M$)**.** The thaw operation above runs the full model's prefill on the decoded tokens, paying standard prefill compute cost on cold reactivation. A more ambitious target is a learned function $\mathcal{U}_M$: tokens $\to$ all-layer KV state that approximates M's native KV-state computation within the depth-adaptive quantization tolerance of Theorem 19. We state this as Conjecture 35 in Section 12, grounded in LAWS-style certified experts [3]: by Sampling the PLT on actual production workflows—"Learning from Actual Workflows Symbolically"—and certifying experts that approximate M's behavior on the resulting PLT-bounded regions, those experts compose into a $\mathcal{U}_M$ for cached content. If $\mathcal{U}_M$ exists with $\sim 10\times$ smaller compute than M, cold-tier thaw latency drops from $\sim$ 100–500 ms to $\sim$ 15–70 ms. The storage compression result of this paper requires no such conjecture; the thaw mechanism above is fully implementable today.

10.7 Storage Algorithm

The compress/decompress pair instantiates as follows in the cache trie:

(1) **Cache write (async).** When segment σ is first computed, store its KV state in the standard hot pool. Schedule asynchronous compression: run E_M on τ_σ to produce the compressed bitstring. Once produced, evict the raw KV state from the hot pool and retain only the compressed representation in cold storage.

(2) **Cache read (sync, parallel).** On a request needing σ, check the hot pool first. If present, return the raw KV state. If only the compressed representation is present, run the layer-0 materialization pipeline (Section 10.5). Promote to hot pool for subsequent reuses if the access pattern justifies it.

(3) **Pinning.** High-traffic segments stay in the hot pool with raw KV. Low-traffic segments compressed to bitstring form. The cache trie's access statistics drive the pinning policy; segments hit more than k times per second stay raw, segments hit less than once per minute compress.

10.8 Storage Estimates

The composition of arithmetic coding against the PLT with the upstream compression stack and structural amortization, for a Llama-3-70B-class model serving 4000-token sessions:

Table 3: Per-session storage at each compression level (70B model, 4000-token session).

Representation	Per session
Raw FP16 KV	1.28 GB
+ Q4 quantization	320 MB
+ KV Sequential delta coding	20 MB
+ PLT-arithmetic encoded ($\sim$ 3 bits/token)	1.5 KB

The arithmetic-coded representation is essentially independent of model size, because it encodes the token sequence rather than the KV state. The per-session ratio over raw FP16 is approximately 850,000× for a 70B model. This is the cross-entropy gap between "raw FP16 KV per token" and "per-token surprisal under the model"—six orders of magnitude of redundancy that exists because the KV vector is a deterministic function of tokens-plus-weights, and the weights are already known to the decoder.

Deployment-scale example. A multi-tenant deployment with 1 platform, 50 communities, 500 bots, and 100,000 sessions:

The full deployment's cache state fits in 150 MB—a $\sim$ 850,000× deployment-level compression, with two factors composing multiplicatively: structural amortization (shared bases stored once per deployment rather than once per session) and arithmetic coding (per-segment storage bounded by source entropy rather than KV size).

Table 4: Deployment-level cache state, 100,000-session multi-tenant deployment.

Layer	Raw FP16	Full PLT stack
1 platform base	1.28 GB	1.5 KB
50 communities	64 GB	75 KB
500 bots	640 GB	750 KB
100,000 sessions	128 TB	150 MB
Total	$\sim$ 128 **TB**	$\sim$ 150 **MB**

10.9 Conservative, Realistic, and Optimistic Operating Points

The compression ratio depends on three factors: (a) the perplexity of the source content under P_M, (b) the perplexity gap between the distilled $\tilde{M}$ and the target M, and (c) arithmetic coding overhead from finite-precision implementation. We give three operating points, each grounded in empirical anchors from the neural-compression literature [20–22].

Table 5: Compression operating points for a 70B model, 4000-token session.

Regime	Bits/token	Per-session	Ratio over raw
Conservative	5.0	2.5 KB	5×10^5
Realistic	3.0	1.5 KB	8.5×10^5
Optimistic	1.7	0.85 KB	1.5×10^6

The conservative bound (5 bits/token). Assumes $\tilde{M}$ has a $\sim$ 50% perplexity gap vs M and the source contains substantial out-of-distribution content (proper nouns, domain-specific terms, technical jargon). Even at this conservative level, per-session storage is 2.5 KB and the compression ratio over raw FP16 KV exceeds 500,000×.

The realistic bound (3 bits/token). Anchored to empirical results in the neural-compression literature. DeepMind's "Language Modeling is Compression" [20] demonstrates that Chinchilla 70B achieves 0.664 bpb on enwik9 via arithmetic coding, which at typical BPE tokenization ($\sim$ 4 bytes/token on English) corresponds to $\sim$ 2.66 bits/token. LLMZip [21] reports 0.71 bits/character for Llama-based compression, also in the $\sim$ 3 bits/token range after tokenization. Modern small-model compressors (Nacrith with SmolLM2-135M, FineZip, ts_zip) achieve similar results at much lower decoder cost [22]. We expect a 1–2B distilled $\tilde{M}$ to operate in this regime on conversational content.

The optimistic bound (1.7 bits/token). Assumes $\tilde{M}$ closely tracks M and the content is highly predictable (system prompts, recurring templates, structured outputs). Below 2 bits/token is achievable for very-in-distribution content where the model assigns near-deterministic probabilities to the source.

Derivation from the theorems. The compression bound is exactly the statement of Proposition 27: $\mathbb{E}[|E_M(\tau_\sigma)|] \leq n_\sigma \log_2 \mathrm{PP}(M) + O(1)$. The arithmetic-coded representation cannot exceed this bound by more than constant overhead, by Shannon's source coding theorem. The data processing inequality (Section 10) further establishes that no per-vector KV compression can be tighter, because the token sequence is a sufficient statistic for the KV state given fixed model weights. The Asymptotic Per-Tenant Storage Theorem (Theorem 14) then composes the structural amortization with the per-segment arithmetic-coding bound, giving the deployment-level numbers above. These are not engineering estimates; they are the consequence of three composed theorems.

10.10 Latency Trade-Off

The architecture has three latency tiers for cache access:

- *Hot raw KV.* Already materialized in GPU HBM or unified memory. Access latency: ~ 1 μs.
- *Warm quantized KV.* Quantized KV (Q4–Q8) resident in DRAM, dequantized on access. Access latency: ~ 0.1–1 ms.
- *Cold compressed.* Bitstring on SSD, thawed by running one parallel prefill pass on the decoded tokens. Access latency: $\sim$ 100–500 ms.

Thaw once, reuse indefinitely. The cold-tier thaw cost is paid once per session reactivation, after which the materialized KV state is back in the hot tier and reused without further thaw cost for the entire duration of the session. Combined with the structural amortization of Theorem 14, the thawed segment is also available for reuse across any other sessions whose towers reference it via cache trie sharing. For a base segment used by thousands of sessions, the thaw cost amortizes over thousands of subsequent reads.

Why this is operationally fine. A $\sim$ 100–500 ms latency on session reactivation is below the threshold of interactive frustration for a user resuming an idle conversation. The user perceives reactivation as "opening the app and waiting briefly," not as broken latency. For background or batch deployments where reactivation is asynchronous (an agent waking up to a scheduled task, a user clicking on a saved conversation), the cost is even less noticeable. The storage-to-compute trade is favorable in essentially all multi-tenant deployment scenarios.

10.11 Composition with Centroid Sharing

The arithmetic-coding scheme of this section composes with the centroid-plus-residual approach as follows: rather than encoding each segment's full token sequence against the model's unconditional distribution P_M, encode it against the *centroid-conditional distribution* $P_M(\cdot \mid \text{centroid})$. The conditional distribution has lower entropy (the centroid carries information about the cluster), so the encoded length is shorter.

This is the bridge to Section 11: cross-tenant PLT clustering uses the same arithmetic-coding machinery, but with the centroid context as the conditioning input. The compression ratio improves with cluster quality (small radius means small conditional entropy means short codewords).

11 DEEP COMPRESSION II: CROSS-TENANT PLT CLUSTERING

The asymptotic-storage theorem (Theorem 14) bounds storage when segments are byte-identical across tenants. In practice, communities and bots in a multi-tenant deployment have prompts that are semantically similar but byte-different—"you are a helpful assistant for X," "you are a Socratic tutor in Y," and so on. PLT clustering generalizes byte-identical sharing to PLT-proximate sharing.

11.1 The Clustering Operation

Definition 31 (PLT cluster)**.** A PLT *cluster* of radius r centered at prefix s^* is the set $\{s : d_T(s, s^*) \leq r\}$. Within a cluster, all members have pairwise trie distance bounded by $2r$.

Given a population of community-level prompts $\{B^{(c)}_{\text{sess}}\}_{c=1}^{C}$, we cluster them by trie proximity: identify cluster centers and assign each prompt to its nearest center. Clusters with small radius represent semantically coherent groups (e.g., "customer support" prompts, "creative writing" prompts).

Theorem 32 (Cross-Tenant Cluster Compression)**.** *Let $\{B^{(c)}_{sess}\}_{c=1}^{C}$ be a population of community-level prompts clustered into K groups with average cluster radius $\bar{r}$ (in trie units, equivalent to surprisal bits). Storage of all C prompts via centroid-plus-centroid-conditioned-encoding (Section 10) is*

$$C_{cluster} = K \cdot |\mathrm{KV}_{centroid}| + C \cdot \bar{r} \cdot \textit{bits}.$$

The per-community amortized cost is $C_{cluster}/C = (K/C) \cdot |\mathrm{KV}_{centroid}| + \bar{r}$ bits, which approaches the bounded $\bar{r}$-bits term as $C \to \infty$.

Proof. Each cluster's centroid is stored once at full KV size. Each non-centroid member is encoded by the centroid-conditioned arithmetic coder E^{cond}_M from Section 10, with expected codeword length bounded by the cluster radius $\bar{r}$ in bits. The amortized cost follows immediately. □

Corollary 33 (Stacking with Theorem 14)**.** *When PLT clustering with centroid-conditioned encoding is applied at each level of the cache trie (platform, community, bot), the per-session storage cost approaches*

$$\frac{C_{total}}{N} \to \bar{r}_{dyn} \cdot \textit{bits per session},$$

where $\bar{r}_{dyn}$ is the average conditional surprisal of dynamic content given the segment-chain centroid context. For typical conversational dynamic content, this is in the bytes-to-kilobytes range per session rather than the megabytes range of raw KV storage.

11.2 Realistic Numbers with PLT Clustering

Applying Theorem 32 to a typical multi-tenant deployment: 50 communities with B_{sess} prompts that cluster into 5 semantic groups of average radius $\bar{r} \approx 100$ trie units (the within-cluster surprisal). At 4 bits per trie unit for natural-language conversational content:

- Without PLT clustering (Towers + KV Seq + Q6): B_{sess} at ∼380 KB per session amortized
- With PLT clustering (Theorem 32): residual encoding at $\sim\bar{r} \cdot 4/8 = 50$ bytes per community delta, plus 5 centroids of ≈ 1 MB amortized over 5,000 sessions = ≈ 1 KB per session

Total B_{sess} contribution per session drops from ≈ 380 KB to ≈ 1 KB, a $\approx 350\times$ reduction on this layer.

11.3 Implementation

Clustering operates offline on segment hashes. At deployment time:

(1) Compute the PLT-trie distance matrix among all community B_{sess} prompts. This requires sampling P_M for the trie metric, but is a one-time offline cost.
(2) Run a clustering algorithm (k-medoids, hierarchical clustering with a trie-distance metric) to identify cluster centers and assignments.
(3) For each cluster, compute the centroid's KV state once via full prefill. Store as a level-1 centroid in the cache trie.
(4) For each community, compute the residual KV state $\text{KV}(B^{(c)}_{\text{sess}}) - \text{KV}(\text{centroid}_{k(c)})$ where $k(c)$ is community c's cluster. Store as a residual segment.
(5) At inference time, materialize the tower by retrieving the centroid and adding the community's residual, then proceed with bot-level and dynamic content as usual.

The clustering algorithm and trie-distance sampling are described in the PLT paper [1]. The KV residual encoding is described in KV Sequential [2]. This section composes them at the cache architecture level.

12 DEEP COMPRESSION III: UNIVERSAL PLT COMPRESSION

The clustering of Section 11 is per-deployment: each multi-tenant operator runs the clustering algorithm on its own prompts and stores its own centroids. The deepest compression target is universal: ship a single set of PLT-derived structures with the model itself, applicable to every deployment of that model.

12.1 Model-Derived Structural Compression

Definition 34 (Universal cache decoder)**.** For model M with L layers, a *universal cache decoder* $\mathcal{U}_M : \text{tokens} \to \text{KV}^{(1:L)}$ is a learned function that maps a token sequence directly to the full all-layer KV state that M would produce on that sequence. The decoder is derived once from model weights and ships with the model.

Conjecture 35 (Universal cache decoder existence). *For any transformer M, there exists an efficient $\mathcal{U}_M$ (with compute cost much smaller than M itself) such that for any input token sequence τ:*

$$\left\|\mathcal{U}_M(\tau) - \text{KV}_M^{(1:L)}(\tau)\right\|_2 \le \varepsilon_\ell \text{ per layer } \ell,$$

where the per-layer error budget ε_ℓ is bounded by the depth-adaptive quantization tolerance (Theorem 19). The approximation error preserves attention output to within the same tolerance the architecture already accepts for quantized cache representations.

LAWS-grounded plausibility argument. The plausibility of Conjecture 35 rests directly on the LAWS framework [3]: "Learning from Actual Workflows Symbolically." LAWS samples the PLT on actual production workflows—real user prompts, real conversation patterns, real RAG contexts—and certifies experts that approximate M's behavior on the resulting PLT-bounded regions. For a multi-tenant chatbot deployment, the cached content is exactly the kind of distribution LAWS targets: highly structured, recurrent, drawn from a narrow slice of the model's full input space.

Where LAWS provides certified experts for substitution at inference time, the same machinery can produce $\mathcal{U}_M$ for cache reconstruction at thaw time. The construction:

(1) Sample actual workflows from the production deployment to identify high-probability PLT regions.
(2) For each region, train a small expert that approximates M's all-layer KV computation on that region. The training target is straightforward: ground-truth KV state from M, supervised regression.
(3) At thaw time, the runner identifies the PLT region of the cached content, dispatches to the corresponding expert, and obtains an approximate KV state within the depth-adaptive quantization tolerance.

The empirical anchor that smaller networks can approximate larger networks' behavior is established by speculative decoding [23]: small draft models routinely achieve $> 70\%$ acceptance rate at 5–10× compute reduction. $\mathcal{U}_M$ is a closely related approximation problem; the LAWS PLT-sampling approach gives the structural prior that makes the approximation tractable.

Quantized PLT-sampling models. The distilled $\tilde{M}$ already shipping with the model can serve as the basis for $\mathcal{U}_M$'s training. At Q4 quantization, a 1B-parameter $\tilde{M}$ samples the PLT cheaply; the same model, with auxiliary output heads for per-layer KV state, becomes $\mathcal{U}_M$. The deployment-time cost is one small-model forward pass on the cached tokens, producing an approximate KV state ready for use.

12.2 Practical Path: Centroid Library

Even without solving Conjecture 35, the centroid-plus-residual approach of Section 10 can be made universal. Define

a fixed library of PLT centroids for each model, covering the high-probability regions of $T(M)$. Ship this library with the model.

The centroid library is sized to cover the empirical distribution of LLM use cases: chatbot system prompts, RAG document patterns, code generation contexts, instruction-following templates. For each centroid, the model's precomputed KV state at layer 0 is stored alongside model weights.

When a deployment caches a segment, the runner identifies the segment's nearest centroid in the library, stores the residual encoding (small, bounded by intra-cluster radius), and reconstructs at inference time by adding the residual to the library centroid.

Proposition 36 (Universal library size)**.** *A centroid library of K entries, each at $|\mathrm{KV}_{centroid}|$ bytes, provides universal compression to within $\bar{r}$ trie units per segment where $\bar{r}$ is the maximum library-centroid-to-real-prompt distance. The library size is fixed per model (a few GB for a 70B model with $K \sim 10^3$ centroids), shipped once.*

12.3 The Limit Behavior

If Conjecture 35 holds, the limit of compression is the information-theoretic minimum: per-session storage is the Kolmogorov complexity of the session's content relative to the PLT prior. For typical conversational content, this is bytes per turn, not megabytes per session.

The asymptotic per-tenant theorem (Theorem 14) said per-session storage $\to |B_{\mathrm{dyn}}|$ at $\Theta(L)$ bytes per dynamic token. The universal-PLT bound replaces this with per-session storage $\to H(B_{\mathrm{dyn}})/8$ bytes per session, where H is the dynamic content's entropy under the model's prior. For dynamic content with high PLT-prior probability (typical conversational replies), this is $\approx$ 100–500 bytes per session, not 1 GB.

12.4 Operational Implication

The MacBook scenario (Section 8.3) with universal PLT compression: the 4 TB SSD holds millions of sessions, not hundreds of thousands. The 64 GB unified memory holds the centroid library plus a working set of recent residuals. Every transformer model shipped with a PLT-coordinate decoder enables this for free at the deployment site.

The architectural shift is significant: cache state stops being a per-deployment artifact and becomes a per-model artifact, like model weights themselves. The cost of a new session is the cost of its incremental information content under the model's prior, which is the information-theoretic minimum.

12.5 Open Questions

The universal-PLT vision raises several open problems:

- *Decoder construction.* Is there an efficient procedure for deriving $\mathcal{U}_M$ from model weights? The PLT itself is not enumerable; sampling-based approximations may suffice for the centroid library approach, but a full universal decoder requires a more principled construction.
- *Centroid selection.* For the library approach, what is the optimal centroid distribution? Equally-spaced trie-distance centroids? Frequency-weighted by P_M? Adaptive to observed deployment statistics?
- *Cross-model transfer.* Are PLT structures shareable across model families with similar architectures (e.g., Llama 70B and Llama 405B)? If so, the universal library can be a foundation-model-level artifact, not a per-checkpoint one.
- *Composition with model substitution.* LAWS [3] replaces model inference with certified experts at PLT nodes. The universal PLT cache compression and LAWS substitution operate on the same structural resource (PLT nodes) and should compose, but the exact composition rule is open.

13 CONCLUSION

The KV cache memory wall has been treated as a per-session problem with per-session solutions. We have shown that for multi-tenant deployments with hierarchical prompt structure, the cache wall is in fact much lower—approaching Shannon's lower bound for the source.

Three stages of compression, stacked. The basic Towers architecture (Sections 5–7) amortizes shared base segments across tenants, driving per-session storage from $\Theta(L)$ toward $|B_{\mathrm{dyn}}|$ as deployment size grows. Depth-adaptive quantization (Section 6) shrinks base segments further with bounded attention error. Arithmetic coding against the PLT (Section 10)—using a small distilled language model to estimate the source distribution—encodes each segment in surprisal bits rather than KV bytes, an additional 10,000× reduction per segment. The thaw mechanism (Section 10.5) decompresses a cold segment by parallel arithmetic decoding followed by one parallel prefill pass, paying $\sim$ 100–500 ms per session reactivation and then handing the materialized KV state back to the standard cache trie where it is reused without further thaw cost.

The entropy-floor result. For a 70B-class model serving typical conversational content, per-segment storage drops from $\sim$ 320 KB per token (raw FP16) to $\sim$ 3 bits per token (arithmetic-coded against P_M). Combined with multi-tenant amortization, per-session storage at 4000-token sessions is approximately 1.5 KB—about 850,000× smaller than the 1.28 GB raw FP16 footprint, within the perplexity-bit envelope of the source distribution. The cache state stops being a per-session megabyte-to-gigabyte-scale artifact and becomes a per-session bitstring whose size is determined by the conversation's surprisal, not its token count.

Position in the literature. The modular attention reuse mechanism we use is well-established prior art, beginning with PromptCache [11] and extended by a substantial follow-up literature [7, 12–19]. Our contribution is not a new caching mechanism but a theoretical and architectural framework:

asymptotic scaling analysis at the deployment level; depth-adaptive quantization with bounded error; deployment patterns spanning four orders of magnitude of hardware scale; and arithmetic coding against the PLT as the source-coded representation that closes the gap to the source-coding lower bound.

The headline operational result. A 64 GB MacBook with a 4 TB SSD, running a 70B model with the full compression stack, holds approximately 2.7 billion sessions on its internal storage at $\sim$ 1.5 KB per compressed session. The decoder bundle ($\sim$ 500 MB for the distilled $\tilde{M}$) occupies less than a gigabyte of unified memory permanently. Approximately 15 GB remains for the hot pool after model weights (40 GB Q4), decoder, and OS overhead—more than enough for tens of thousands of warm sessions. Cold-tier reactivation runs one parallel prefill pass per thaw ($\sim$ 100–500 ms), after which the segment is back in the hot cache for unlimited subsequent reuse and cross-session sharing through the cache trie. Per-session storage is bounded by the entropy of the conversation under the model, not by the conversation's token count. A personal computer becomes a fully production-quality platform for multi-tenant LLM deployment at any scale a single developer or small team would plausibly need to host.

The scaling continuum. The same architecture deploys to commodity workstations (32 GB, CPU-only), to discrete-GPU servers with shared-memory swap, and to multi-TB-RAM production servers. The theoretical foundation is the same across all four orders of magnitude of hardware scale. The PLT compression closes the lower-bound gap consistently across all scales; the differences are in raw inference throughput, not in storage feasibility.

An ambitious theoretical extension. If the universal cache decoder $\mathcal{U}_M$ of Conjecture 35 can be built—by training LAWS-style certified experts [3] on PLT regions sampled from actual deployment workflows—cold-tier thaw drops from $\sim$ 100–500 ms to $\sim$ 15–70 ms, an additional order of magnitude. This is not required for the storage-compression result. It is the natural research direction that follows from composing this paper's source-coded cache architecture with the LAWS framework for certified substitution. The conjecture is grounded in the empirical observation that small models routinely approximate large models on bounded distributions (speculative decoding shows 5–10$\times$ compute reduction), and the PLT-sampling approach of LAWS is the structural prior that makes the approximation tractable on cached content specifically.

The architecture is implementable in stages. Stock llama.cpp for the compute-savings hierarchical caching; a delta-extraction utility ($\sim$ 500 lines of C++) for the asymptotic storage scaling; shared-memory swap or unified-memory tower assembly for the platform-specific GPU paths; and the arithmetic coding codec (with the distilled probability model $\tilde{M}$) for the entropy-floor compression. The arithmetic-coding codec is the most ambitious immediately-deployable piece; it ships once with each model and benefits every deployment.

Combined with the prior PLT-based compression program [1–3], this paper completes a four-layer compression stack covering structural amortization, sequential entropy, layered quantization, and source-coded representation. The KV cache memory wall, treated rigorously as an information-theoretic problem, has a floor much lower than per-session analysis suggests—and the architecture to reach it fits in a personal laptop.

REFERENCES

[1] G. Magarshak. Probabilistic language tries: A unified framework for compression, decision policies, and execution reuse. *arXiv preprint arXiv:2604.06228*, 2026.

[2] G. Magarshak. Sequential KV cache compression via probabilistic language tries: Beyond the per-vector Shannon limit. *arXiv preprint arXiv:2604.15356*, 2026.

[3] G. Magarshak. LAWS: Learning from actual workloads symbolically—a self-certifying parametrized cache architecture for neural inference, robotics, and edge deployment. *arXiv preprint arXiv:2605.04069*, 2026.

[4] G. Magarshak. Grokers: Bottom-up inductive comprehension and write-time intelligence over typed knowledge graphs. *arXiv preprint*, 2026.

[5] A. Zandieh, M. Daliri, M. Hadian, V. Mirrokni, P. Kacham, L. Gottesburen, and R. Jayaram. TurboQuant: Online vector quantization with near-optimal distortion rate. In *International Conference on Learning Representations (ICLR)*, 2026.

[6] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles*, 2023.

[7] L. Zheng et al. SGLang: Efficient execution of structured language model programs. *arXiv preprint arXiv:2312.07104*, 2024.

[8] J. Su, Y. Lu, S. Pan, A. Murtadha, B. Wen, and Y. Liu. RoFormer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024.

[9] G. Xiao, Y. Tian, B. Chen, S. Han, and M. Lewis. Efficient streaming language models with attention sinks. In *International Conference on Learning Representations (ICLR)*, 2024.

[10] S. Chen, S. Wong, L. Chen, and Y. Tian. Extending context window of large language models via positional interpolation. *arXiv preprint arXiv:2306.15595*, 2023.

[11] I. Gim, G. Chen, S.-S. Lee, N. Sarda, A. Khandelwal, and L. Zhong. Prompt Cache: Modular attention reuse for low-latency inference. In *Proceedings of Machine Learning and Systems (MLSys)*, 2024.

[12] J. Yao, H. Li, Y. Liu, S. Hu, Y. Cheng, Q. Zhang, K. Du, S. Lu, J. Jiang, and H. Hoffmann. CacheBlend: Fast large language model serving for RAG with cached knowledge fusion. In *Proceedings of EuroSys*, 2025.

[13] S. Lu, Y. Wang, M. Sheng, B. Liu, and Y. Zhang. TurboRAG: Accelerating retrieval-augmented generation with precomputed KV caches for chunked text. *arXiv preprint*, 2024.

[14] J. Yang, Z. Hu, W. Liu, B. Zhao, and C. Cheng. KVLink: Accelerating large language models via efficient KV cache reuse. *arXiv preprint arXiv:2502.16002*, 2025.

[15] L. Ye, Z. Tao, Y. Huang, and Y. Li. ChunkAttention: Efficient self-attention with prefix-aware KV cache and two-phase partition. *arXiv preprint arXiv:2402.15220*, 2024.

[16] Y. Liu, H. Li, K. Du, J. Yao, Y. Cheng, S. Lu, M. Maire, H. Hoffmann, A. Holtzman, G. Ananthanarayanan, and J. Jiang. LMCache: An efficient KV cache layer for enterprise-scale LLM inference. *arXiv preprint arXiv:2510.09665*, 2025.

[17] Z. Bian, F. Wu, C. Zhang, H. Dong, Y. Liang, and Y. Zhuo. TokenDance: Scaling multi-agent LLM serving via collective KV cache sharing. *arXiv preprint arXiv:2604.03143*, 2026.

[18] Y. Yang, M. Liu, S. Zhang, and J. Wang. KVShare: An LLM service system with efficient and effective multi-tenant KV cache reuse. *arXiv preprint arXiv:2503.16525*, 2025.

[19] W. Chen, Z. Liu, X. Sun, and L. Wang. SemShareKV: Efficient KVCache sharing for semantically similar prompts via token-level LSH matching. *arXiv preprint arXiv:2509.24832*, 2025.

[20] G. Delétang, A. Ruoss, P.-A. Duquenne, E. Catt, T. Genewein, C. Mattern, J. Grau-Moya, L. K. Wenliang, M. Aitchison, L. Orseau, M. Hutter, and J. Veness. Language modeling is compression. In *International Conference on Learning Representations (ICLR)*, 2024.

[21] C. S. K. Valmeekam, K. Narayanan, D. Kalathil, J.-F. Chamberland, and S. Shakkottai. LLMZip: Lossless text compression using large language models. *arXiv preprint arXiv:2306.04050*, 2023.

[22] R. Tacconelli. Nacrith: Neural lossless compression via ensemble context modeling and high-precision CDF coding. *arXiv preprint arXiv:2602.19626*, 2026.

[23] Y. Leviathan, M. Kalman, and Y. Matias. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning (ICML)*, 2023.