High-Throughput LLM-Based Compression and What It Takes

A Systems Study of Inference Engines and Entropy Coding Optimizations

David Adams
University of Melbourne
Melbourne, Australia
david.adams1@student.unimelb.edu.au

Luisa Neubauer
University of Technology Nuremberg
Nuremberg, Germany
luisa.neubauer@utn.de

Ping-Lin Kuo
University of Technology Nuremberg
Nuremberg, Germany
ping-lin.kuo@utn.de

Mihail Stoian
University of Technology Nuremberg
Nuremberg, Germany
mihail.stoian@utn.de

Thomas Rückstieß
Relaxed Constraints
Melbourne, Australia
thomas@relcon.ai

Josif Grabocka
University of Technology Nuremberg
Nuremberg, Germany
josif.grabocka@utn.de

Renata Borovica-Gajic
University of Melbourne
Melbourne, Australia
renata.borovica@unimelb.edu.au

Andreas Kipf
University of Technology Nuremberg
Nuremberg, Germany
andreas.kipf@utn.de

Abstract

Large language models are attractive text compressors because next-token probabilities can be converted into code lengths. In practice, however, the main bottleneck is often not modeling quality but systems cost: a compressor must obtain logits quickly enough to compete with strong classical baselines.

This paper examines systems optimizations for LLM-based compression, including multistream pipelined arithmetic coding and target-interval capture, and evaluates their trade-offs across inference engines, configurations and datasets. We evaluate LLM-based text compression on three different inference engines: Transformers, vLLM, and TensorRT-LLM. Notably, we obtain a speedup of 18.84x compared to prior work's Transformers implementation and a maximum recorded inference speed of 181 kilobytes per second at a compression ratio of 7.17, when running on an NVIDIA A100.

VLDB Workshop Reference Format:
David Adams, Luisa Neubauer, Ping-Lin Kuo, Mihail Stoian, Thomas Rückstieß, Josif Grabocka, Renata Borovica-Gajic, and Andreas Kipf. High-Throughput LLM-Based Compression and What It Takes. VLDB 2026 Workshop: Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures.

1 Introduction

Real-world analytical database systems store and process substantial amounts of textual data. In an analysis of the Amazon Redshift fleet, `varchar` columns account for 52.1% of stored columns, making variable-length strings the most prominent datatype in the observed workloads [21]. Similarly, an analysis of production Snowflake workloads reports that text columns account for 34% of aggregation inputs, substantially more than other non-numeric types [17]. These findings suggest that efficient storage and processing of textual data are relevant concerns for modern analytical database systems.

Compression is particularly attractive for textual data that is retained but accessed or modified infrequently. The Redshift study reports that 86.6% of fleet tables received no inserts or deletes during the monitored period, and identifies compression as a potential optimization for such data [21]. It also finds that half of the queries reading external data sources target compressed files, nearly all of which use gzip, demonstrating that compression already forms part of production analytical data pipelines. For cold text data, stronger compression could reduce storage and data movement costs, provided that it does not impose prohibitive costs when data is ingested or subsequently decoded for analysis.

Large language models (LLMs) offer a new approach to lossless compression of textual data. In LLM-based compression, a pre-trained model supplies next-token probability distributions, which an entropy coder converts into a compact bitstream. While these distributions may enable substantial storage reductions, obtaining them requires neural inference during both compression and decompression. Thus, to be practically useful in accelerating data management systems, LLM-based compression must be evaluated against core Database Management System (DBMS) metrics. Specifically, we must measure the trade-offs between storage footprint reduction, ingestion throughput, and decompression speed during query execution.

Motivation. This motivates our central question: under which combinations of inference engine, hardware platform, and entropy-coding design might LLM-based compression become economically viable as a storage codec for textual data? To investigate this question, we evaluate the systems-level economics of deploying LLM-based compressors across inference engines and their configurations, and entropy coder optimizations, examining their trade-offs and viability alongside standard DB compressors FSST [2] and `zstd` on these crucial dimensions.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment. ISSN 2150-8097.

Related Work. Neural lossless compression treats compression as next-token prediction followed by entropy coding. Early work by Schmidhuber and Heil [16] showed that neural sequence models could be combined with Huffman [6] or arithmetic coding [15] for lossless text compression, establishing the basic connection between neural prediction quality and conventional text compression algorithms. Later systems used deeper neural models to improve compression ratios. DeepZip [4] combines gated RNNs, including LSTMs and bidirectional GRUs, with arithmetic coding and reports compression ratio improvements over gzip on text and genomic data. DecMac [11] similarly uses an LSTM-based context model with cycle connections, reporting further reductions over gzip and 7zip. These systems show that neural predictors can improve on compression ratio, but also make compression substantially more expensive because model training and token-by-token inference dominate runtime.

DZip [5] reduces some of this cost with a hybrid adaptive training strategy, avoiding separate external training data while remaining general-purpose, but still reflects the central trade-off in neural compression: better probability estimates require more computation. TRACE [12] shifts this trade-off toward faster inference by replacing recurrent models with a lightweight transformer architecture. Its single-layer transformer, byte grouping, and shared feed-forward components target the sequential bottlenecks of earlier neural compressors.

Recent work replaces these task-trained predictors with pre-trained large language models. LLMZip [20] demonstrates this idea by using a pre-trained LLaMA-7B [19] model, reporting strong compression ratios on English text. Li et al. [10] extend this argument more broadly, showing that large pre-trained models can serve as powerful compressors across different data modalities.

However, removing training does not remove the systems cost. LLM-based compression still requires repeated next-token probability queries, often in a highly sequential loop. FineZip [14] and Nacrith [18] therefore focus on making LLM-based compression more practical through online memorization, lightweight auxiliary predictors, higher-precision coding, backend optimizations and selective use of the LLM. Kipf et al. [7] explore the trade-offs between compression ratios and latency and make the limitations explicit from a systems perspective: the compression gains of LLMs must be weighed against slow inference and decompression costs.

A parallel line of work studies the problem of serving LLMs efficiently. vLLM [8] improves throughput by managing the key-value cache using PagedAttention, reducing memory waste and enabling larger effective batch sizes. SGLang [24] introduces runtime optimizations such as RadixAttention and efficient structured decoding for programs that make many language model calls. Although these systems were not designed primarily for compression, they are highly relevant because LLM-based arithmetic coding is dominated by repeated probability queries.

Contribution. This work sits at the intersection of neural compression and inference systems. Prior LLM-based compression work argues that stronger predictive models can improve code length, but often treats the cost of obtaining logits as a secondary concern [10, 13, 14, 20].

We also relate to work on high-throughput LLM inference. Engines such as vLLM, Transformers, and TensorRT-LLM focus on different inference optimizations, and those design choices can directly affect whether compression is viable on server GPUs, CPUs, or even edge devices.

Finally, we compare against strong classical baselines such as LZ4 and xz. These compressors remain the practical reference point because they are fast, cheap, and broadly deployed. Our goal is therefore not only to ask whether LLM-based compression can improve compression ratio, but under which system configurations it can do so at acceptable speed and cost [3, 22, 23].

In particular, our optimized implementation on these three inference engines achieves speedups of 18.84x, 1.85x, and 2.16x, respectively, over prior work's baseline implementation [7], when run on a NVIDIA A100, making a step towards high-throughput LLM-based compression.

2 Problem Setting

We implement lossless compression as an append-only next-token prediction workload. Given a tokenized sequence, the compressor repeatedly queries a language model on the current prefix and converts the predicted distribution for the next token into compressed output. This workload makes logit access, prefix reuse, and serving overhead more important than general inference.

2.1 Global-Mask Compression

Let $x_{1:n}$ be the token sequence obtained from the input under the tokenizer of a fixed language model M. The implementation first constructs a global allowed-token set

$$V' = \begin{cases} \{x_i : 1 \le i \le n\} \subset V, & \text{if vocabulary reduction is enabled,} \\ V, & \text{otherwise,} \end{cases}$$

where V is the full tokenizer vocabulary. The set V' is serialized as a bitmap and stored with the compressed payload. All subsequent probability vectors are restricted to this single global vocabulary, rather than using a different candidate set per chunk.

The token stream is split into B contiguous rows of lengths $m_1, \ldots, m_B$, where B is the configured batch size, and the first token of each row is stored verbatim as the seed needed for decoding. At step t, the compressor forms the context

$$c_t = x_{\max(1,\, t-R+1):t},$$

where R is the configured context-length limit. The predictor queries the model for next-token logits

$$z_t = M(c_t) \in \mathbb{R}^{|V|}.$$

For probability-based encodings the logits are restricted to V' and normalized:

$$p_{t+1,\,j} = \frac{\exp z_{t,j}}{\sum_{k \in V'} \exp z_{t,k}}, \qquad j \in V'.$$

Arithmetic coding encodes the true next token x_{t+1} under the probability mass function p_{t+1}, consuming $-\log_2 p_{t+1,\,x_{t+1}}$ bits, so the model's cross-entropy directly determines the ideal compressed size before coding overhead and the bitmap.

The implementation also supports rank-based encodings. In that case the predictor returns logits rather than probabilities, and the compressor records the zero-indexed rank

$$r_{t+1} = \left|\{j \in V' \mid z_{t,j} > z_{t,x_{t+1}}\}\right|.$$

The rank sequence is then encoded with either a fixed-width bit-packed representation or a Huffman code. These encodings measure how often the model places the correct token near the top of the distribution, while avoiding a full probability-to-arithmetic-coder conversion at each step.

Decompression mirrors the same autoregressive process. Given the bitmap, initial row tokens, and compressed payload, the decoder reconstructs V', queries the same model on the reconstructed prefixes, and decodes each next token from the arithmetic stream. The decoded token is appended to the context for the following query. Compression is lossless only when the model, tokenizer, vocabulary mask, context policy, and arithmetic-coding parameters are identical between compression and decompression.

3 Implementations

Due to the nature of modern LLM inference engines and the compression workload, we found it necessary to implement custom logit capture paths for both vLLM and TensorRT-LLM. These paths are not intended for general use, but they allow us to evaluate the trade-offs of different inference engines without confounding factors from separate compressor implementations. The same compressor loop can be used with Hugging Face (HF)'s Transformers, vLLM, or TensorRT-LLM backends, and the same predictor interface can be used for custom logit capture or engine-backed scores.

3.1 Custom and Engine-Backed Logits

The baseline path uses HF's Transformers directly. It pads variable-length prompts when needed, forwards the prompt batch through the model, takes the final-position logits, optionally restricts them to the global token mask, and applies a softmax only for probability-based encodings. With KV caching enabled, the first call builds a cache from the full prompt and subsequent calls feed only the newly appended token until the context is truncated and the cache must be rebuilt. This path is simple and gives explicit control over logits, but each compression step is still a Python-level loop around model execution, score conversion, and entropy coding.

Because vLLM is optimized for text generation, its batch scheduler does not return next-token scores in the row-stable order required for compression. We therefore use a POSIX shared-memory interface: a logits processor runs inside the vLLM worker after the dense logits are computed, copies the selected scores into a pre-allocated capture buffer, and marks each row-step entry with a readiness flag. Each request carries a row identifier so the host can recover the original batch order regardless of vLLM's internal scheduling. When vocabulary reduction is enabled, only the reduced-vocabulary columns are copied, keeping transfer costs proportional to $|V'|$ rather than the full padded vocabulary. Implementation details and earlier design iterations are described in Appendix A.

The TensorRT-LLM path uses the runtime's offline APIs, which expose generation logits directly. The predictor builds or loads a cached engine configured for the compression batch size and context length, issues a one-token generation request per prompt batch, and receives a dense [batch, vocab] tensor of final-position scores. KV-cache management is handled internally by the runtime, so no host-side cache maintenance is required. The same global token mask and deferred softmax used by the other backends are applied before scores are passed to the entropy coder.

3.2 Deployment Variants

We evaluate three practical deployment styles.

Transformers backend. This backend loads the Hugging Face model in the compressor process. It supports LoRA adapters, seq2seq and Mamba variants where configured, and explicit cache-free or cached operation. It is the most portable implementation and serves as the correctness reference for the other engines.

vLLM backend. This backend uses vLLM's offline `LLM.generate` API with prefix caching enabled when requested. It supports tensor parallelism, configurable GPU memory utilisation, and a fixed maximum model length matching the compression context length. Generation is used as the mechanism that causes vLLM to run the model and invoke the logits processor; the generated token is not the primary output.

TensorRT-LLM backend. TensorRT-LLM is integrated through the same predictor interface. It can build or reuse a cached engine directory and return the same reduced probability/logit matrices as the other paths. The implementation currently targets decoder-only models and relies on TensorRT-LLM's internal runtime cache management. This keeps the evaluation axes clean: model, hardware, inference engine, and encoding can vary without changing the compressor loop.

3.3 Optimization

Using unmodified engines with arithmetic coding is not economical [7]. We frame three optimizations not just as compressor tweaks, but as generic acceleration patterns for LLM-as-a-service within databases. These patterns target different parts of the end-to-end critical path: request scheduling inside the inference engine, probability-to-interval conversion for arithmetic coding, and score-tensor transfer back to the host.

Teacher-forced vLLM windows. A naive compressor issues one next-token request per compression step, paying the fixed overhead of request creation, scheduling, and shared-memory synchronisation at every token. Instead, the compressor sends the current prefix together with the next w ground-truth tokens from the input stream. The vLLM logits processor records the score vector at each internal decode step and forces the sampler to emit the known target token, so one engine call collects scores for an entire window. This amortises per-request overhead across w compression steps and allows the KV cache to be reused across successive positions within the window. See Appendix B for compatibility caveats and measured overheads.

Multistream pipelined compiled arithmetic coding. Because each batch row is compressed independently, the implementation maintains one range coder per row and converts probability vectors to target intervals in tensor form, avoiding per-row Python

loops. A Numba-compiled [9] backend executes the integer range-coder update loop over compact arrays of interval triples. On the Transformers backend, interval conversion and encoding for one batch are overlapped with the next model forward pass via a background thread, reducing exposed CPU-side coding latency. Full quantization and pipelining details are given in Appendix B.

Target-interval capture. The arithmetic coder requires only the quantized interval assigned to the true next token, not the full probability vector. In target-interval mode the inference backend computes this interval in place and returns a fixed-size record $(\ell, h, T, p_{\text{target}})$ per row and step. For the vLLM backend this is particularly effective: the logits processor performs the softmax and interval derivation inside the worker and writes only the fixed-size record to shared memory, reducing transfer cost from $O(|V'|)$ floats to $O(1)$ per row-step.

4 Experiments

All experiments use the `Qwen/Qwen2.5-0.5B` model. With the exception of the hardware section, all experiments were run on the same hardware configuration: two AMD EPYC Milan 7313 16-core 3 GHz processors, 512 GB DDR4-3200 ECC memory, and a NVIDIA A100 40 GB SXM4 4-GPU module. The default operating point is a batch size of 256, a context length of 256, and 100 retained tokens from the dataset (as the initial context). We utilize datasets from prior work [7]: (i) the `text8` dataset (100 MiB), an English text extracted from Wikipedia; (ii) Python code, namely the first 100 MiB from PyTorrent [1]; and (iii) the Yelp dataset, containing 150,346 business records from the Yelp Open Dataset.[1] Reported throughput is in MB/s (uncompressed bytes per wall-clock second) and compression ratio as original size divided by compressed size.

4.1 Experiment A: Engine Comparison

Table 1 compares the three inference engines at a fixed batch size and encoding configuration. The transformer model is also evaluated without its KV cache to isolate the cost of the incremental forward pass.

Table 1: Inference engine comparison.

Engine	Variant	(MB/s)	ratio	speedup
Transformers	+ KV cache	0.0344	7.0492	1.00×
	No KV cache	0.0030	7.0502	0.09×
vLLM	Default	0.0130	7.0304	0.37×
TensorRT-LLM	Batch 256	0.0081	7.0499	0.023×
	Batch 512 (max)	0.0065	7.9370	0.019×

The most striking result in Table 1 is that neither of the two production-grade inference engines outperforms the plain Hugging Face's Transformers. Specifically, vLLM reaches only 0.37× the Transformers' throughput and TensorRT-LLM fares worse still at 0.023×. This is not a flaw in the engine but a mismatch between their design objectives and the compression workload. They are designed for high-concurrency serving: they optimize request scheduling, memory pooling, and latency fairness across independent queries. LLM-based AC, by contrast, is a stateful, sequential workload in which every token depends on the one before it and all rows in a batch must be processed in the correct order to feed the entropy coder. As the per-request overhead of this workload is negligible in a serving context the scheduling, shared-memory synchronization and result reassembly becomes the dominant cost incurred by these engines.

[1] Data obtained from `https://www.yelp.com/dataset`.

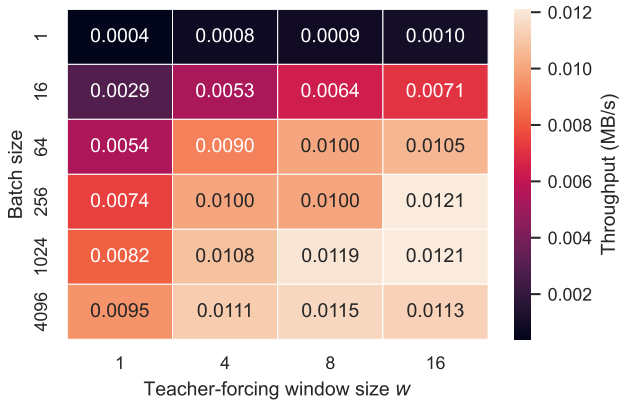

Figure 1: vLLM throughput (MB/s) as a function of both batch size and teacher-forcing window size w (varying O1, using O2). Setting $w = 1$ corresponds to no teacher forcing.

4.2 Experiment B: Optimization Ablation

We evaluate every applicable combination of the three compression-specific optimizations:

- **O1.** Teacher-forced vLLM windows (vLLM only; window size $w = 8$).
- **O2.** Multistream pipelined arithmetic coding (Multistream AC, Numba packed backend and pipeline encoding on HF's Transformers).
- **O3.** Target-interval capture (Target interval AC).

Combinations that are architecturally inapplicable (O1 on transformer or TensorRT-LLM) are marked N/A. Compression ratio is expected to remain constant across all rows within an engine because O2 and O3 are lossless algorithmic equivalents of the baseline.

Table 2 exposes a fundamental bottleneck-shift pattern that differs sharply across engines, and which has direct implications on how each engine should be configured in a database storage stack.

Transformers. At baseline, the Transformers engine spent 97.1% of wall time inside the arithmetic coder. In this case the model forward pass is essentially free by comparison. O2 alone achieves no improvement because without O3 there is still a full probability vector to process per step and the pipeline has nothing useful to overlap. O3 alone eliminates the bottleneck entirely: by computing only the quantized interval for the true next token inside the inference step, the AC time drops from 97.1% to 1.0% and throughput increases by 18.96×. This is the single most impactful optimization across all engines. Adding O2 on top of O3 yields minimal gains

Table 2: Optimization ablation at batch size 256, context length 128, `text8` dataset. Relative speedup is normalized to each engine's own baseline. O1 applies only to vLLM.

Engine	Configuration	(MB/s)	ratio	Rel. speedup	AC time (%)
Transformer	Baseline (AC)	0.0018	7.0482	1.00×	97.1
	+ O2	0.0018	7.0482	1.00×	97.1
	+ O3	0.0347	7.0492	18.96×	1.0
	+ O2 + O3	0.0345	7.0492	18.84×	1.0
vLLM	Baseline (AC)	0.0031	7.03	1.00×	89.9
	+ O1	0.0116	7.05	3.43×	9.2
	+ O2	0.0116	7.05	3.43×	9.2
	+ O3	0.0065	7.03	1.94×	3.0
	+ O1 + O2	0.0118	7.05	3.48×	9.2
	+ O1 + O3	0.0063	7.05	1.87×	2.3
	+ O2 + O3	0.0066	7.03	1.94×	3.1
	+ O1 + O2 + O3	0.0062	7.05	1.85×	2.3
TensorRT-LLM	Baseline (AC)	0.0020	7.0488	1.00×	58.2
	+ O2	0.0020	7.0488	1.00×	58.2
	+ O3	0.0044	7.0499	2.18×	0.7
	+ O2 + O3	0.0043	7.0499	2.16×	0.7

for such small datasets as there is not enough work to adequately distribute across threads.

vLLM. The vLLM baseline is slightly more balanced than the Transformers' 89.9% AC time, but achieves only a slightly higher absolute baseline of 0.0031 MB/s, reflecting the per-request overhead that is always present regardless of coding cost. O1 (teacher forced windows) and O2 produce identical speedups of 3.43× and their AC profiles are also similar. This suggests that both reduce the same cost of issuing one engine call per token but via different mechanisms and they are not additive. O3 alone gives a more modest 1.94× (0.0065 MB/s), because unlike the Transformers case, inference overhead is not negligible in vLLM, so eliminating the coding step does not produce a proportionally large gain.

Crucially, combining all three optimisations (O1+O2+O3) slightly *regresses* to 0.0062 MB/s (1.85×). This counter-intuitive result reflects a scheduling interaction: O1 restructures requests into windows that the vLLM scheduler processes differently, and when O3 is also active the reduced shared-memory transfer per step interacts with the window batching in a way that slightly increases idle time.

TensorRT-LLM. TensorRT-LLM shows a markedly different AC time profile at baseline: 58.2%, indicating that the engine itself already imposes substantial non-coding overhead compared to the plain Transformers. O2 yields no improvement, as with Transformers. O3 delivers a 2.18× speedup (0.0044 MB/s), but this is far smaller than the 18.96× seen on Transformers because a larger fraction of time is already spent in engine overhead that O3 cannot address. This suggests that TensorRT-LLM's internal runtime costs of engine initialization, offline API scheduling, and KV-cache management present a floor that no coding-side optimization can break through. Unless future versions of TensorRT-LLM expose lower-overhead logit access paths for stateful sequential workloads, it is unlikely to be competitive with the Transformers backend for this use case.

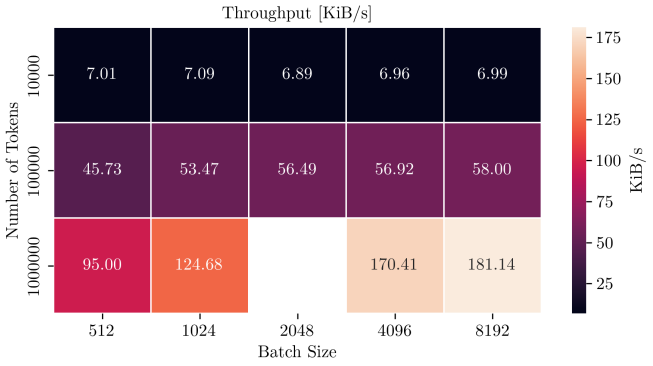

Figure 2: Transformers throughput (Kb/s) as a function of both batch size and number of tokens processed. Missing values are repeating out of memory errors.

4.3 Experiment C: Batch-Size Sweep

We report in Figure 1 the throughput at every batch size for vLLM, making the bottleneck shift visible, and additionally sweeping the teacher-forcing window sizes $w \in \{1, 4, 8, 16\}$ to quantify the O1 × batch interaction. It shows that the benefit of teacher-forcing is most pronounced at small-to-medium batch sizes, where the per-request scheduling overhead dominates. At larger batch sizes the inference step itself becomes more GPU-saturating, and the marginal value of amortizing request overhead across a window of w tokens diminishes. This mirrors a well-known pattern in database query execution: batching reduces fixed per-tuple overheads but offers diminishing returns once the variable per-tuple cost dominates. From an operator's perspective, window size $w = 8$ represents a practical sweet spot: it delivers most of the available gain without the additional memory pressure of larger windows.

Figure 2 shows that processing more tokens improves overall hardware utilization and throughput.

4.4 Experiment D: Context-Length Sensitivity

Longer contexts increase the per-step inference cost and alter KV-cache reuse patterns. Table 3 reports throughput and compression ratio for Transformers at the fixed batch size across context lengths $R \in \{32, 64, 128, 256, 512\}$ with retain length set to $R/2$.

Table 3: Context-length sensitivity on Transformers with the fully optimized encoding path (+ O2 + O3), batch size 256, `text8` dataset. Retain length is set to half the context length in each case. The default operating point (context 256, retain 100) is shown in bold.

Context length	Retain tokens	(MB/s)	ratio
32	16	0.0324	6.2288
64	32	0.0338	6.6592
128	64	0.0339	6.9239
256	**100**	**0.0337**	**7.0600**
512	256	0.0349	7.1007

Table 3 presents a practically important trade-off for storage system designers. Throughput is nearly flat across the range 64-512 tokens (0.0338-0.0349 MB/s), varying by less than 3%. Compression ratio, however, improves materially with context length: from 6.23 at context 32 to 7.10 at context 512. The improvement is concave: the largest gains occur between context lengths 32 and 128, with diminishing returns thereafter consistent with the model's ability to exploit local lexical and syntactic patterns within the first hundred tokens, beyond which marginal predictive value from additional history decreases.

This decoupling of throughput from context length is a favorable property from a systems perspective. A storage administrator can increase context length to improve the compression ratio of cold data reducing storage footprint and data movement costs at essentially no throughput penalty, provided GPU memory is sufficient to hold the enlarged KV cache. At context 512 the KV cache is twice as large as at context 256 but delivers only a 0.7% throughput improvement (7.10 vs. 7.06 ratio), making 256 the more memory-efficient default.

4.5 Experiment E: Encoding Quality

Table 4 reports compression quality for each encoding scheme on the text8 dataset with the Transformers backend at the default operating point. Arithmetic-coding variants (AC, Multistream AC, Target interval AC) and PMATIC are lossless-optimal up to quantisation; rank-based schemes (bitpacked, Huffman) trade compression ratio for coding simplicity. Classical baselines are included for reference.

Table 4 places the LLM-based encodings in context against alternative coding schemes and classical compressors. Among the arithmetic-coding variants, Target interval AC (O3) achieves the best ratio (7.1007) at 0.0349 MB/s, marginally improving on Multistream AC (7.0492 at 0.0345 MB/s). The difference in bits/token 5.1201 vs. 5.1642 is small but consistent, reflecting that target-interval capture loses slightly less to quantisation by operating directly on the coder interval rather than on a rounded probability vector.

Table 4: Compression quality on `text8`, Transformers engine, batch size 256, context length 256. Bits/token is the empirical cross-entropy; classical baselines do not use an LLM so bits/token is not applicable.

Encoding	(MB/s)	ratio	Bits/token
AC (Baseline)	0.0359	6.1962	0.0000
Multistream AC (O2)	0.0345	7.0492	5.1642
Target interval AC (O3)	0.0349	7.1007	5.1201
PMATIC	timeout	–	–
Bitpacked	0.0363	2.8735	N/A
Huffman	0.0359	6.1962	N/A
Classical baselines (CPU, no LLM)			
LZ4	666.66	1.83	N/A
xz -9e	1.34	4.30	N/A
zstd	586.70	3.06	N/A
FSST	294.11	2.08	N/A

Table 5: Compression quality on different datasets, fully optimized Transformers engine (+ O2 + O3), batch size 256, context length 256.

Dataset	(MB/s)	ratio
`text8`	0.005436	7.739
`code`	0.004132	11.203
`Yelp`	0.002490	7.028

The rank-based schemes tell a cautionary tale. Bitpacked encoding retains the throughput of arithmetic coding but collapses the ocmpression ratio to 2.87, worse than zstd (3.06) at a fraction of the cost. Huffman encoding matches AC in tatio but offers no throughput advantage. PMATIC's timeout is notable: it represents an approach that attempts higher coding precision at the cost of per-token latency that is not practical.

4.6 Discussion of DB-Centric Compressors

When placing these DB-centric classical baselines (e.g., zstd, FSST) on the throughput-ratio cost Pareto frontier, we observe that they heavily prioritize ingestion throughput and decompression speed. While LLM-based compression achieves unprecedented compression ratios by accurately modeling the data distribution, its throughput remains orders of magnitude slower, fundamentally bottlenecked by the model's forward pass. In scenarios querying cold storage where storage cost dominates, LLM compression may be viable. However, for active query paths and high-throughput ingestion, lightweight compressors like FSST and zstd currently provide a far more economical balance.

4.7 Experiment F: Encoding Backend Ablation

This experiment holds the inference engine (Transformers with KV cache) and batch size fixed while varying only the arithmetic-coding backend, thread count, and pipelining flag. The goal is to

Table 6: Backend × pipeline ablation for O2 (Multistream AC), Transformers engine, batch size 256, `text8` dataset.

Backend	Threads	Pipeline	(MB/s)	AC (%)	speedup
Raw python	–	off	0.0008	36	1×
numpy	–	off	0.0914	13.3	57.01×
numba_threaded	auto	off	0.0920	9.3	57.42×
numba_threaded	4	off	0.0935	9.7	58.36×
numba_threaded	1	off	0.0940	9.4	58.67×
numba_packed	–	off	0.0949	9.2	59.22×
numba_threaded	auto	on	0.4847	48.9	302.48×
numba_threaded	4	on	0.4860	49.1	303.29×
numpy	–	on	0.4867	71.0	303.71×
numba_packed	–	on	0.4879	47.1	304.45×
numba_threaded	1	on	0.4879	49.2	304.49×

Table 7: Backend × pipeline ablation for O3 (Target interval AC), Transformers engine, batch size 256, `text8` dataset.

Backend	Threads	Pipeline	(MB/s)	AC (%)	speedup
Raw python	–	off	0.0008	36	1×
numpy	–	off	0.0908	13.4	56.63×
numba_threaded	4	off	0.0936	9.6	58.42×
numba_threaded	auto	off	0.0938	9.4	58.52×
numba_threaded	1	off	0.0940	9.4	58.65×
numba_packed	–	off	0.0951	9.1	59.35×
numba_threaded	1	on	0.4808	48.4	300.05×
numpy	–	on	0.4813	71.2	300.34×
numba_threaded	4	on	0.4836	48.6	301.77×
numba_threaded	auto	on	0.4851	48.8	302.69×
numba_packed	–	on	0.4853	46.5	302.85×

isolate coding overhead from inference cost and to guide backend selection across hardware targets.

The four Numba-capable encodings (AC, Multistream AC, Target interval AC) are each tested with the `python`, `numba`, `numba_threaded`, and `numba_packed` backends. For threaded backends, thread counts of 1, 4, and auto (0, which selects the number of CPU cores) are included. For `Multistream AC` and `Target interval AC`, pipelining (`–pipeline_encoding`) is tested both on and off. Rank-based encodings (`bitpacked`, `huffman`) and `PMATIC` are included as single reference rows since they do not expose a backend choice.

Table 6 covers Multistream AC; and Table 7 covers Target interval AC

Tables 6 and 7 isolate the arithmetic coding layer from the inference engine and reveal two distinct operating regimes separated by the pipelining flag.

Without pipelining. The jump from a simple python implementation of AC with cumulative multiplications to a vectorized numpy implementation is the largest single step: 57× for Multistream AC. This reflects the cost of per-token Python interpreter overhad in the inner coding loop: at batch size 256 there are 256 independent range coders to update per step which is trivially parallelisable in numpy. Numba just in time compilation offers further gains of 2-4% over numpy. At this scale threading provides no benefit: numba_threaded at 1, 4, and auto threads all land within 1% of each other (0.0920–0.0940 MB/s for Multistream AC). This indicates that the bottleneck without pipelining is not thread-level parallelism but the memory bandwidth of updating 256 compact interval arrays, a pattern that saturates a single core.

Enabling the pipeline flag causes a further ∼5× jump across all backends, converging to 0.48–0.49 MB/s regardless of whether numpy, numba, or numba_packed is used. The tight clustering spread across all pipelined backbends show that pipelineing the active coding step is fully overlapped with model inference and the remaining AC time fraction (41-71%) show that the inference is to short to fully cover the remining work.

The practical implication here is that the choice of using numba matters only when pipelining is unavailable e.g a CPU-only deployment. In such a regime numba_packed is the best single choice. When pipelining is available, numpy suffices and avoids the just in time compilation overhead at startup, which may be relevant for short-lived compression jobs.

With pipelining. Enabling the pipeline flag causes a further ∼5× jump across all backends, converging to 0.48–0.49 MB/s regardless of whether numpy, numba, or numba_packed is used. The tight clustering shows that with pipelining active the coding step is fully overlapped with model inference and the remaining AC time fraction (47–71%) reflects coding work that cannot be hidden because the inference step is too short to cover it. The Numpy backend shows the highest AC time fraction (71.0%) at comparable throughput, suggesting slightly less efficient overlap than Numba backends, but the absolute difference in throughput is negligible.

The practical implication is the choice of Numba variant matters only when pipelining is unavailable (e.g., on a CPU-only deployment): in this regime, numba_packed is the best single choice. When pipelining is available, Numpy suffices and avoids the just in time compilation overhead at startup, which may be relevant for short-lived compression jobs.

4.8 Reproducibility Notes

To assess run-to-run variability, we repeated a subset of throughput measurements (Transformers engine, no optimizations, varying batch sizes and context lengths) five times per configuration and found low variation, with a mean coefficient of variation at 0.76%. Due to limited compute resources, we therefore report single-run measurements for the remaining experiments.

5 Discussion: Throughput Bottleneck Analysis

The end-to-end compressor has three potential bottlenecks: inference latency, arithmetic-coding pre-processing, and score-extraction overhead. The dominant bottleneck shifts as batch size and engine optimizations change.

Low-batch regime. At small batch sizes, throughput is determined by model inference latency. The compressor stalls after each token step waiting for the next score vector, so faster engines such as vLLM and TensorRT-LLM directly raise throughput by shortening that wait. GPU utilization is low in this regime because fixed launch costs dominate and the arithmetic-coding side is idle most of the time.

Arithmetic-coding pre-processing at larger batches. As batch size grows, the arithmetic-coding side becomes the bottleneck. The baseline AC path requires probability normalization, conversion to integer frequencies, and cumulative-prefix construction for every active row at every compression step $O(|V'|)$ work repeated across the full batch. This overwhelms the savings from larger GPU batches and caps useful parallelism.

Effect of multistreamed arithmetic coding and pipelining. The multistream backend reduces this cost in two ways. First, it eliminates full cumulative-vector construction per token event: target intervals are staged compactly and fed directly to row-local range coders. Second, the Numba-compiled backends move the hot renormalization loop out of Python and, in the packed variant, write byte-oriented streams directly without materializing intermediate bit lists. When pipelining is enabled on the transformer backend, interval staging and encoding are overlapped with the next model forward pass. Together these changes remove enough exposed coding latency that larger GPU batch sizes become useful again, shifting the bottleneck back toward inference.

Score-extraction overhead. Once arithmetic-coding cost is reduced, the next bottleneck for the vLLM and transformer paths is score extraction. Both still require a dense logit or reduced-vocabulary probability matrix to reach the host at every step unless target-interval capture is active. For vLLM, this transfer is particularly expensive because logits must traverse shared memory out of the worker process, and that cost can dominate short next-token requests. Target-interval capture mitigates this by reducing the exported payload to a constant-size record per row-step, but dense-logit capture remains costly when full distributions are needed.

Transformer backend after optimization. Despite being less specialized than TensorRT-LLM and less throughput-oriented than vLLM, the transformer backend is the most flexible path for this workload: it exposes logits directly, supports host-managed control flow, and can overlap arithmetic coding with the next forward pass without shared-memory capture hooks. As a result, once the AC path is optimized, the transformer backend can match or exceed the other engines on end-to-end compression throughput even when those engines are faster for ordinary text serving. This reflects a broader point: the compression workload has different systems requirements from generation, and inference engines designed with direct interval or reduced-vocabulary capture in mind could reduce all three bottlenecks simultaneously.

6 Conclusion

This paper evaluated the systems-level economics of LLM-based lossless text compression across inference engines, entropy-coding optimisations, and hardware configurations. While LLMs consistently achieve strong compression ratios over 7× on natural language and 11× on code, the key question is whether they can do so economically compared to classical compressors.

Our results show that the main bottleneck is not model inference but arithmetic coding. High-throughput inference engines such as vLLM and TensorRT-LLM perform poorly in this setting because their scheduling strategies are mismatched to the sequential token-by-token nature of arithmetic coding. In the baseline transformer pipeline, 97% of runtime is spent in the entropy coder. The target-interval capture optimisation (O3) removes this bottleneck, yielding an 18.96× speedup and shifting runtime dominance entirely to the model forward pass.

These findings highlight that systems engineering matters as much as model quality. For a fixed model, throughput can vary by nearly two orders of magnitude depending on implementation. Techniques such as teacher-forced windows, pipelined multistream coding, and target-interval capture generalise beyond compression to other stateful sequential LLM workloads.

Despite these gains, LLM-based compression remains several orders of magnitude slower than LZ4 and zstd, making it impractical for latency-sensitive or high-throughput workloads. Its strongest use case today is cold, write-once data where storage reduction outweighs compute cost. The code dataset results 11.2× compression versus zstd's 3.06× demonstrate the potential value for suitable data types.

Future work should focus primarily on reducing inference cost through smaller or specialised models, speculative decoding, and improved GPU scheduling. Since entropy coding is no longer the dominant bottleneck after optimisation, further gains are likely to come from faster inference rather than coding improvements.

Overall, LLM-based compression is not yet a replacement for classical codecs, but with careful systems engineering it is becoming practical for cold-storage analytical workloads where maximum compression ratio is more valuable than throughput.

References

[1] Mehdi Bahrami, N. C. Shrikanth, Shade Ruangwan, Lei Liu, Yuji Mizobuchi, Masahiro Fukuyori, Wei-Peng Chen, Kazuki Munakata, and Tim Menzies. 2021. PyTorrent: A Python Library Corpus for Large-scale Language Models. arXiv:2110.01710 [cs.SE] https://arxiv.org/abs/2110.01710

[2] Peter A. Boncz, Thomas Neumann, and Viktor Leis. 2020. FSST: Fast Random Access String Compression. *Proc. VLDB Endow.* 13, 11 (2020), 2649–2661. http://www.vldb.org/pvldb/vol13/p2649-boncz.pdf

[3] Peter A. Boncz, Thomas Neumann, and Viktor Leis. 2020. FSST: Fast Random Access String Compression. *Proc. VLDB Endow.* 13, 11 (2020), 2649–2661.

[4] Mohit Goyal, Kedar Tatwawadi, Shubham Chandak, and Idoia Ochoa. 2019. DeepZip: Lossless Data Compression Using Recurrent Neural Networks. In *2019 Data Compression Conference (DCC)* (2019-03). 575–575. doi:10.1109/DCC.2019.00087

[5] Mohit Goyal, Kedar Tatwawadi, Shubham Chandak, and Idoia Ochoa. 2021. DZip: Improved General-Purpose Loss Less Compression Based on Novel Neural Network Modeling. In *2021 Data Compression Conference (DCC)* (2021-03). 153–162. doi:10.1109/DCC50243.2021.00023

[6] David A. Huffman. 1952. A Method for the Construction of Minimum-Redundancy Codes. 11 (1952), 91–99. https://api.semanticscholar.org/CorpusID:10606404

[7] Andreas Kipf, Tobias Schmidt, Ping-Lin Kuo, Skander Krid, Moritz Rengert, Luca Heller, Andreas Zimmerer, Mihail Stoian, Varun Pandey, and Alexander van Renen. 2026. Waiting to Decompress: The Economics of LLM-Based Compression. In *Conference on Innovative Data Systems Research.* https://api.semanticscholar.org/CorpusID:284161408

[8] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23).* Association for Computing Machinery, New York, NY, USA, 611–626. doi:10.1145/3600006.3613165

[9] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. 2015. Numba: a LLVM-based Python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC.* 1–6. doi:10.1145/2833157.2833162

[10] Ziguang Li, Chao Huang, Xuliang Wang, Haibo Hu, Cole Wyeth, Dongbo Bu, Quan Yu, Wen Gao, Xingwu Liu, and Ming Li. 2025. Lossless data compression by large models. *Nature Machine Intelligence* 7, 5 (2025), 754–799. doi:10.1038/s42256-025-01033-7

[11] Qian Liu, Yiling Xu, and Zhu Li. 2019. DecMac: A Deep Context Model for High Efficiency Arithmetic Coding. In *2019 International Conference on Artificial Intelligence in Information and Communication (ICAIIC)* (2019-02). 438–443. doi:10.1109/ICAIIC.2019.8668843

[12] Yu Mao, Yufei Cui, Tei-Wei Kuo, and Chun Jason Xue. 2022. TRACE: A Fast Transformer-based General-Purpose Lossless Compressor. In *Proceedings of the ACM Web Conference 2022* (New York, NY, USA, 2022-04-25) *(WWW '22)*. Association for Computing Machinery, 1829–1838. doi:10.1145/3485447.3511987

[13] Yu Mao, Holger Pirk, and Chun Jason Xue. 2025. Lossless Compression of Large Language Model-Generated Text via Next-Token Prediction. *CoRR* abs/2505.06297 (2025). arXiv:2505.06297 doi:10.48550/ARXIV.2505.06297

[14] Fazal Mittu, Yihuan Bu, Akshat Gupta, Ashok Devireddy, Alp Eren Ozdarendeli, Anant Singh, and Gopala Anumanchipalli. 2024. FineZip : Pushing the Limits of Large Language Models for Practical Lossless Text Compression. *CoRR* abs/2409.17141 (2024). arXiv:2409.17141 doi:10.48550/ARXIV.2409.17141

[15] J. Rissanen and G. G. Langdon. 1979. Arithmetic Coding. 23, 2 (1979), 149–162. doi:10.1147/rd.232.0149

[16] J. Schmidhuber and S. Heil. 1996. Sequential Neural Text Compression. 7, 1 (1996), 142–146. doi:10.1109/72.478398

[17] Jan Vincent Szlang, Sebastian Bress, Sebastian Cattes, Jonathan Dees, Florian Funke, Max Heimel, Michel Oleynik, Ismail Oukid, and Tobias Maltenberger. 2025. Workload Insights from the Snowflake Data Cloud: What Do Production Analytic Queries Really Look Like? 18, 12 (2025), 5126–5138. doi:10.14778/3750601.3750632

[18] Roberto Tacconelli. 2026. Nacrith: Neural Lossless Compression via Ensemble Context Modeling and High-Precision CDF Coding. arXiv:2602.19626 [cs.IT] doi:10.48550/arXiv.2602.19626

[19] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. https://arxiv.org/abs/2302.13971v1

[20] Chandra Shekhara Kaushik Valmeekam, Krishna Narayanan, Dileep Kalathil, Jean-François Chamberland, and Srinivas Shakkottai. 2023. LLMZip: Lossless Text Compression using Large Language Models. *CoRR* abs/2306.04050 (2023). arXiv:2306.04050 doi:10.48550/ARXIV.2306.04050

[21] Alexander Van Renen, Dominik Horn, Pascal Pfeil, Kapil Vaidya, Wenjian Dong, Murali Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf, and Tim Kraska. 2024. Why TPC Is Not Enough: An Analysis of the Amazon Redshift Fleet. 17, 11 (2024), 3694–3706. doi:10.14778/3681954.3682031

[22] Lukas Vogel, Alexander van Renen, Satoshi Imamura, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2020. Mosaic: A Budget-Conscious Storage Engine for Relational Database Systems. *Proc. VLDB Endow.* 13, 11 (2020), 2662–2675.

[23] Leon Windheuser, Christoph Anneser, Huanchen Zhang, Thomas Neumann, and Alfons Kemper. 2024. Adaptive Compression for Databases. In *EDBT*. OpenProceedings.org, 143–149.

[24] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2024. SGLang: Efficient Execution of Structured Language Model Programs. In *Proceedings of the 38th International Conference on Neural Information Processing Systems (NeurIPS '24, Vol. 37)*. 62557–62583.

A Engine Backend Implementation Details

A.1 vLLM Logits Extraction

Since vLLM does not return next-token scores in a row-stable order suitable for compression, early versions used `prompt_logprobs`'s output for rank-based encodings and experimented with bulk rank extraction from that interface. This is sufficient for bitpacked and Huffman variants, where only the rank of the true token in the sorted score list is needed, but arithmetic coding requires a full distribution over the reduced vocabulary at every step, making top-k logprob APIs insufficient.

We also evaluated vLLM's `logits_processors` interface with score capture to temporary files. This confirmed that vLLM can expose the required scores, but the file handoff introduced latency that made the path uneconomical for the compression workload.

The current implementation uses POSIX shared memory. The host process allocates a capture buffer and passes its name through environment variables. The logits processor attaches to that buffer, copies the selected logits as `float32`, and marks completed row-step entries with readiness flags. Each request carries a row identifier in vLLM's sampling parameters so the host can recover the original batch order even when vLLM reorders requests internally.

A.2 TensorRT-LLM Tensor Extraction

TensorRT-LLM can return logits in several payload formats and with engine-specific padding in the vocabulary dimension. The predictor extracts the last valid time step from the returned tensors, trims any padded vocabulary columns back to the semantic tokenizer vocabulary, and then applies the global token mask. We prefer lower-level model-runner interfaces when available and fall back to `LLM.generate` otherwise; both are normalised to the same [batch, vocab] contract.

B Optimization Implementation Details

B.1 Teacher-Forced vLLM Windows

The logits processor forces the sampler to emit the known target token at each internal decode step by setting all other logits to $-\infty$. The window size w is bounded by the configured context length and the number of retained context tokens so that every captured score vector corresponds exactly to the prefix the decoder will reconstruct.

In our experience this optimization is not compatible with vLLM's tensor parallelism and is therefore restricted to single-GPU configurations. Even at large batch sizes, the overhead of POSIX shared-memory synchronisation remains higher than the transformer path. The optimization nonetheless illustrates how the compression workload can benefit from custom scheduling and data-movement strategies inside the inference engine.

B.2 Multistream Arithmetic Coding

Probabilities are normalized in `float64`, clamped to non-negative values, and scaled to a fixed total $T = 2^{18}$ with a minimum count of one per symbol. The target interval (ℓ, h, T) is extracted from row-wise prefix sums without constructing a per-row cumulative table on the host.

The Numba-compiled [9] backend implements the standard integer range-coder update loop with renormalization and underflow handling, operating on compact arrays of interval triples rather than Python objects. A packed variant writes bits directly into byte arrays and returns an explicit bit count, avoiding the memory overhead of materializing Python lists. Row streams are independent, so final encoding can be parallelised across CPU threads.

On the transformer backend, interval conversion and batch encoding for step t are submitted to a background thread immediately after the probability batch is produced. The main thread begins step $t{+}1$ without waiting and synchronises only if the background task for step t has not yet completed, overlapping GPU execution with CPU-side coding latency.

B.3 Target-Interval Capture

In target-interval mode the inference backend reduces the vocabulary if requested, computes the softmax locally, derives the interval for the forced token, and writes only a fixed-size record $(\ell, h, T, p_{\text{target}})$ into shared memory. This changes the data-transfer cost from $O(|V'|)$ floats per row-step to $O(1)$, greatly reducing PCIe and shared-memory traffic and removing the host-side cumulative-table construction step for the transformer and TensorRT-LLM paths.