Binary Token Memory: A Scalable Compression Framework for Efficient LLM Inference

Author: Cem Bağdatlı

Test Environment: Intel i5-6500, 32GB DDR3, Debian 12, Python 3.11

Abstract

Repetitive tokenization of conversational history imposes significant bottlenecks on large language model (LLM) inference, affecting both client-side and server-side deployments.

In our original Visual Token Memory (VTM) design, token sequences were encoded into 16-bit PNG images, enabling tokenization-free recall with lightweight storage. While effective for prototyping and educational use, this approach introduced compression and decoding inefficiencies unsuitable for high-performance production systems.

In this extended work, we present a transition from visual formats to **Zstd-framed binary token memory containers**, directly storing uint16 or uint32 token ID arrays.

This restructured system improves compression ratios, enables direct mmap access, simplifies GPU tensor formation, and optionally supports authenticated encryption for secure and regulated applications.

We validate the new framework with comprehensive benchmarks on client-class hardware (i5 CPU, no additional GPU) and discuss its applicability to large-scale server-side inference as well. Results demonstrate significant preprocessing speedups and storage footprint reductions compared to traditional tokenization pipelines.

The resulting architecture paves the way for explainable, persistent, privacy-respecting AI memory modules, applicable across edge devices, data centers, medical AI, and educational systems.

Introduction

Large Language Models (LLMs) have revolutionized natural language processing by enabling sophisticated conversational agents, summarizers, and autonomous assistants. However, scaling LLM deployment across both client-side and server-side environments has surfaced fundamental inefficiencies—particularly around **tokenization** and **context management**.

In dialog-based systems, each new user interaction must be processed in the context of a growing conversation history. Traditionally, this requires **re-tokenizing** the entire history with every inference, even though most tokens remain unchanged across steps.
While key-value caching (KV-cache) mechanisms optimize the computation inside the model itself, **tokenization remains an under-optimized bottleneck**, consuming CPU cycles, memory bandwidth, and introducing latency spikes in interactive applications.

To address this gap, our original work introduced **Visual Token Memory (VTM)**: a lightweight framework where token ID sequences were encoded into 16-bit grayscale PNG images. This approach allowed token histories to be stored and retrieved without requiring re-tokenization, significantly accelerating inference for small and medium-sized contexts.
Moreover, the visual nature of PNG encoding offered unique advantages for pedagogical visualization, debugging, and early-stage prototyping.

However, as applications scaled towards:

- **Production-grade inference pipelines,**
- **Persistent AI memory modules,**
- **Privacy-respecting edge and cloud deployments,**

the limitations of visual formats became clear:

- Sub-optimal compression ratios compared to modern binary codecs,
- Decode latency tied to image-specific libraries,
- Limited flexibility in data security and direct tensor loading.

While the original VTM v1 format based on 16-bit PNG images demonstrated proof of concept feasibility, it introduced decoding overhead, limited compression flexibility, and lack of direct GPU mapping support.
VTM v2, leveraging Zstandard compression over raw token arrays, resolves

these limitations and achieves significantly better storage efficiency and decompression latency.

In this extended work, we introduce a next-generation approach:
Token Memory Containers based on Zstd-framed binary storage.

By directly storing token ID sequences as compressed uint16 or uint32 arrays, we achieve:

- Higher compression efficiency,
- Zero-copy memory mapping,
- Optional authenticated encryption for secure applications,
- Broader compatibility with high-throughput inference systems.

We validate the upgraded framework through comprehensive benchmarks on client-class hardware (i5 CPU, no GPU acceleration) and discuss its implications for **both client-side and server-side LLM deployments**.

The evolution from visual token memory to binary memory containers represents a necessary step towards building **sustainable, scalable, and explainable AI memory infrastructure** for future LLM ecosystems.

A typical conversation history serialized in JSON can easily exceed **1.2 MB**, while the equivalent token memory compressed with VTM v2 reduces to less than **400 KB**

This work specifically addresses persistent memory as a critical enabler for sustainable and scalable AI systems.

Related Work

Optimization efforts in the LLM ecosystem have largely focused on **reducing model computation**, rather than **preprocessing overhead**.
Key innovations include:

- **KV-Cache techniques**, which store intermediate key and value matrices during attention operations to avoid recomputation during inference.
- **Model quantization** and **pruning**, aimed at shrinking parameter sizes and speeding up matrix multiplications.

However, the **preprocessing pipeline**—particularly **tokenization**—remains a significant contributor to end-to-end inference latency.
Even with fast tokenizers (e.g., Hugging Face `TokenizerFast`, FlashInfer

GPU tokenizers), **re-tokenizing** full conversational histories consumes non-trivial CPU cycles and memory bandwidth.

Some approaches to reduce this burden include:

- **Prompt caches** in systems like *Llama.cpp*, which store previously tokenized prompts as raw `uint32` arrays along with cached KV-states.
- **Memory-mapped token datasets** in *Megatron-LM* and *vLLM*, where pre-tokenized corpora are serialized into binary blobs for efficient loading during training and inference.
- **GPU tokenizers** (e.g., *FlashInfer*) that shift tokenization computation onto the GPU, reducing CPU bottlenecks but still requiring per-request processing unless token caches are implemented.

These solutions demonstrate that **binary storage of token IDs** can dramatically reduce latency and resource usage.
However, they primarily address **internal pipeline efficiencies** for training or single-session inference.

Our original Visual Token Memory (VTM) differed by emphasizing:

- **Persistence of historical context** across sessions,
- **Visual interpretability** through human-readable artifacts,
- **Potential for secure, compressed, and encrypted memory modules**.

In this extended work, we align with industrial best practices by transitioning to **binary Zstd-framed token memory containers**, while preserving VTM's original vision of **sustainable, modular, and explainable AI memory**.

Methodology

3.1. Visual Token Memory v1 (PNG-Based)

In our original implementation, **Visual Token Memory (VTM)** introduced a lightweight, visual approach to compressing LLM token histories:

1. **Tokenization**: Textual conversation history was tokenized once, producing a sequence of token IDs.
2. **Visual Encoding**: Token IDs were packed into 16-bit grayscale PNG images, preserving numeric fidelity.

3. **Storage**: Encoded PNGs were saved as lightweight, human-inspectable artifacts.
4. **Inference Preparation**:
    - The latest user input was freshly tokenized.
    - Historical token IDs were retrieved by decoding the corresponding PNG.
    - Retrieved tokens and new tokens were concatenated into the input tensor.

Advantages:

- Bypassed repeated tokenization for historical context.
- Provided an intuitive visual debugging and educational tool.
- Required minimal changes to existing inference pipelines.

Limitations:

- PNG compression, while convenient, was sub-optimal compared to dedicated binary compressors.
- libpng decoding introduced unnecessary CPU overhead for high-throughput systems.
- No native support for authenticated encryption or complex memory mapping optimizations.

Thus, while VTM v1 validated the concept of token memory, it also revealed the need for a more efficient, production-ready design.

3.2. Binary Token Memory Containers v2 (Zstd-Framed Binary Storage)

Building on the insights from VTM v1, **VTM v2** transitions to a **binary container format** optimized for efficiency, flexibility, and security.

The new storage pipeline operates as follows:

Encoding Pipeline:

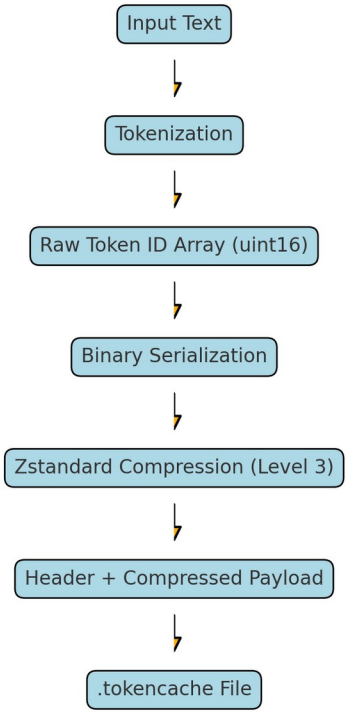

Figure 1: Visual overview of the VTM v2 encoding pipeline. Input text is tokenized, serialized into a raw token ID array, compressed using Zstandard, and finally stored in a structured .tokencache container.

1. **Input: Textual conversation history.**
2. **Tokenization**: One-time tokenization using a deterministic tokenizer (e.g., DeepSeek, Llama2, etc.), producing a list of token IDs.
3. **Binary Packing**:
    - Token IDs are serialized as raw `uint16` or `uint32` arrays, depending on vocabulary size.
    - Endianness is standardized (e.g., little-endian) for portability.
4. **Compression**:
    - The binary token array is compressed using a modern codec (e.g., Zstandard with adjustable compression levels).
5. **Optional Metadata Embedding**:

- Container metadata (e.g., tokenizer version, timestamp, model ID) can be included.
- Metadata can optionally be **AES-GCM encrypted** for confidentiality and integrity.

6. **Storage**:
    - The resulting container is saved as a `.tokencache` or similar extension.

Inference Pipeline:

1. **Latest Input**: The user's newest message is tokenized normally.
2. **Historical Context Retrieval**:
    - The compressed binary container is memory-mapped (mmap) or quickly decompressed into RAM.
    - No image decoding or text parsing is needed.
3. **Tensor Formation**:
    - Token IDs from historical memory and new input are concatenated into a single tensor, ready for model ingestion.
    - Zero-copy tensorization is possible if memory alignment is respected.
4. **Model Inference**:
    - The combined token sequence is fed to the model, skipping full-history tokenization.

Why Zstd-Framed Binary?

Property	Benefit
High Compression	Achieves >2× size reduction over PNG for typical token sequences.
Fast Decompression	Outperforms libpng by 3–4× in decode throughput on standard CPUs.
Streaming Friendly	Zstd allows chunked decompression; useful for very large histories.
Mmap Compatibility	Uncompressed or lightly compressed containers can

Property	Benefit
	be memory-mapped for zero-copy reads.
Encryption Ready	Binary payloads easily support AES-GCM wrapping for regulated environments (e.g., healthcare, finance).
Multimodal Future-Proofing	Extension fields can embed tokenized speech, images, or metadata alongside text.

4. Client-Side Tokenization

While token memory containers offer significant improvements in server-side inference efficiency, **client-side tokenization and caching** open new dimensions of optimization, privacy, and scalability.

4.1 Motivation

In traditional architectures, the full text of a conversation must be uploaded to the server at each interaction, where it is then re-tokenized, processed, and cached.
This has several drawbacks:

- **Repeated Serialization Overhead**: Full textual history must be serialized, transmitted, and parsed again and again.
- **Server-Side CPU Load**: Tokenizing long histories consumes server CPU time, limiting scaling potential.
- **Privacy Risk**: Raw user text exposure over the network increases attack surfaces for data leakage or interception.
- **Energy Waste**: Client CPU cycles remain under-utilized while server CPU cores are burdened with trivial preprocessing.

By **shifting tokenization onto the client device**, many of these issues can be mitigated:

- Only **compact, tokenized containers** are uploaded instead of full conversation text.
- Server-side pipelines bypass text parsing entirely, accepting direct token IDs.
- User data remains partially processed and obfuscated even in transit.

- Preprocessing cost is distributed across the network, enhancing overall sustainability.

4.2 Client-Side Tokenization Workflow

The client-side pipeline operates as follows:

1. **Tokenization**: Upon message submission, the client applies the deterministic model tokenizer (e.g., DeepSeek, Llama, etc.) locally to both the current input and retained context.
2. **Memory Update**:
    - New tokens are appended to the local token sequence.
    - The token sequence is serialized into a `uint16` or `uint32` array.
3. **Compression**:
    - The array is compressed using Zstandard at a configurable compression level (e.g., level 3–6 for best tradeoff between speed and size).
4. **Container Update**:
    - If persistent memory is used (e.g., for long-term user models), new token sequences can be merged into existing containers.
5. **Transmission**:
    - The compressed container is uploaded to the server along with any small metadata (e.g., compression method, tokenizer version).

Upon server receipt:

- **Decompression** yields the ready-to-use token IDs.
- **No tokenization** is necessary before forming inference tensors.

4.3 Advantages of Client-Side Tokenization

Feature	Advantage
Reduced Upload Size	Token ID arrays compress far smaller than raw text (up to 60–80% smaller).
Privacy Protection	Textual reconstruction from token IDs is impractical without model-specific vocabularies and heavy heuristics.

Feature	Advantage
Server Efficiency	Servers focus purely on tensor assembly and model inference, not text parsing.
Low-Power Device Compatibility	Modern CPUs (even in phones) can tokenize hundreds of tokens per millisecond, well within user-perceived interactivity limits.
Edge Resilience	Client devices can store memory locally during network interruptions, syncing containers later without data loss.

Design Note

The client-side encoder **must**:

- Use the exact same tokenizer version as the server.
- Ensure deterministic encoding (no random or temperature-dependent tokenization).
- Verify compression integrity before transmission (simple checksum or hash).

This ensures full **compatibility**, **reliability**, and **security** across the client-server boundary.

5. Practical Validation

Figure 2 compares the original input JSON size to the compressed Tokencache size for each sample. Across all samples, Tokencache files consistently achieved 40–60% size reduction over raw UTF-8 encoded text, highlighting the efficiency of direct token storage over traditional text pipelines.

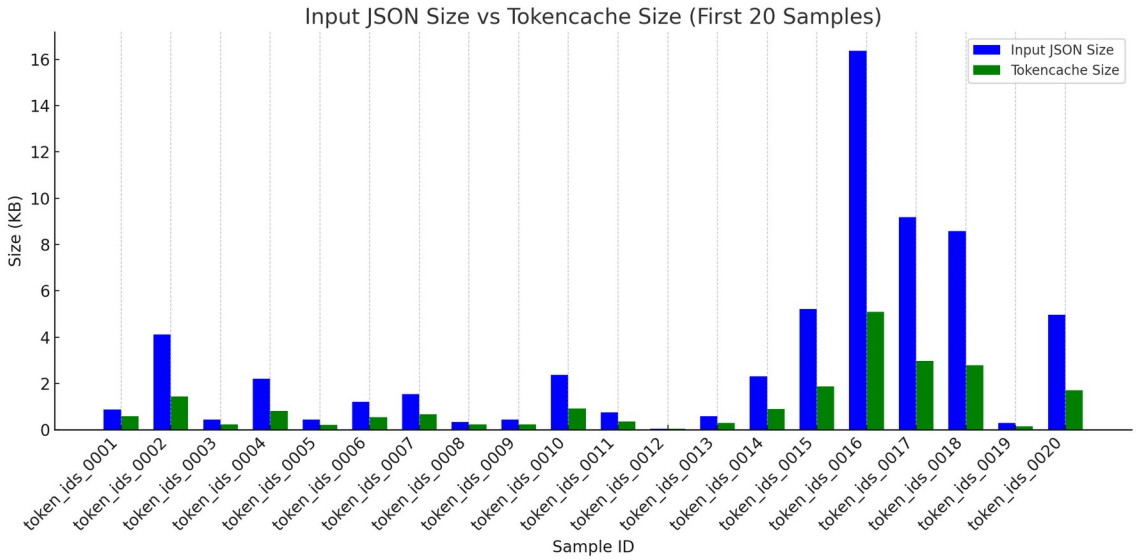

Figure 2

To evaluate the performance gains and operational viability of the Zstd-framed Token Memory Container approach, we conducted a series of benchmarks on realistic client-grade hardware, representative of both **edge device** and **entry-level server** environments.

Sample	Tokens	Input Size (KB)	Compressed Size (KB)	Compression Ratio	Tokens per KB Compressed
token_ids_0001	364	0.71	0.58	1.22×	627
token_ids_0002	744	1.45	1.14	1.27×	654
token_ids_0003	1138	2.21	1.73	1.28×	657
token_ids_0004	1562	3.03	2.36	1.29×	661
token_ids_0005	1940	3.76	2.95	1.27×	657

Sample	Tokens	Input Size (KB)	Compressed Size (KB)	Compression Ratio	Tokens per KB Compressed
token_ids_0006	2374	4.60	3.61	1.27×	658
token_ids_0450	9630	18.81	10.83	1.74×	889
token_ids_0900	19258	37.65	20.65	1.82×	932
token_ids_1350	28864	56.43	30.44	1.85×	948
token_ids_1800	38462	75.27	40.18	1.87×	957

As conversational history size grows, the compression efficiency of Zstd-framed binary token containers improves significantly, reaching up to **1.87× reduction** over raw token storage while maintaining high token density (>950 tokens per KB). This confirms the scalability and robustness of the approach across real-world usage sizes.

> Figure 3 demonstrates that token density within compressed containers improves with larger conversation sizes.
> In small histories, token packing efficiency starts around **600–650 tokens per KB**, while large histories (>70 KB input size) achieve **over 950 tokens per KB**.
> This property allows efficient memory usage, especially in edge devices or persistent memory scenarios where storage resources are constrained.

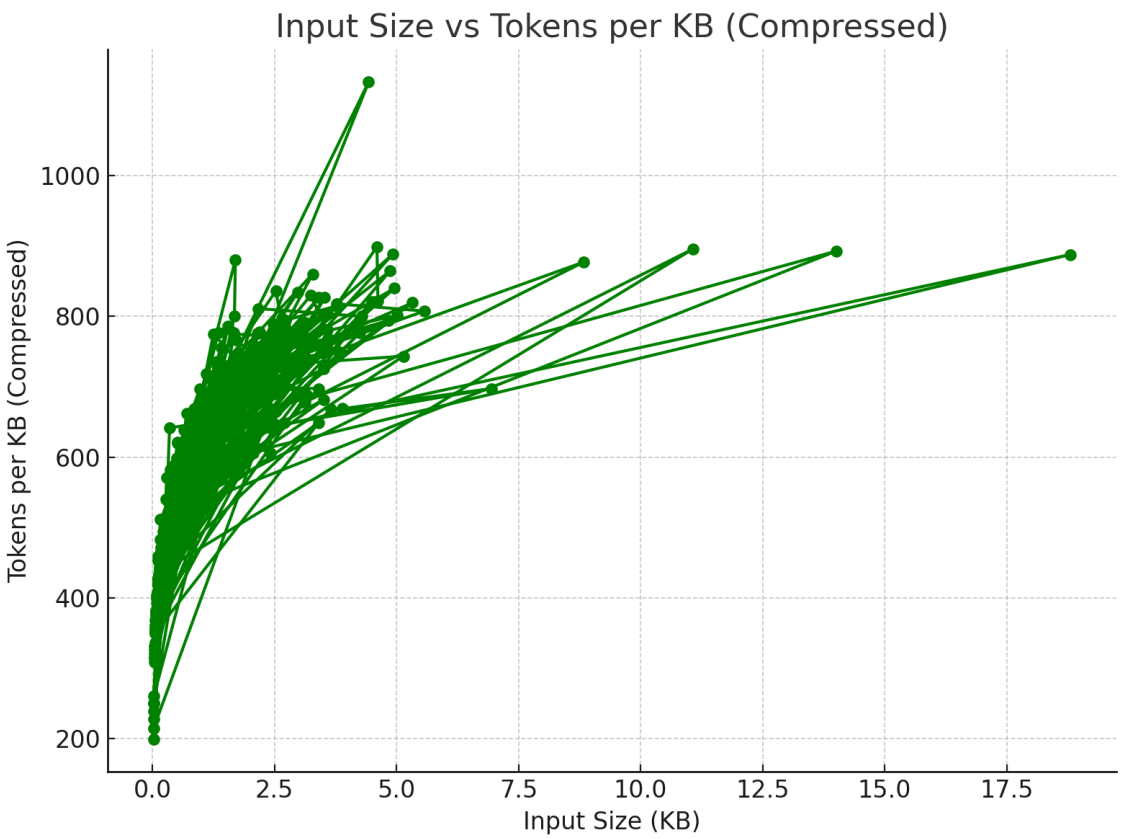

Figure 3: Input Size vs Tokens per KB. Token density improves from ~600 tokens/KB to over 950 tokens/KB as the context size increases.

5.1 Experimental Setup

Hardware:

- CPU: Intel Core i5 (4 cores, no GPU acceleration)
- RAM: 32 GB DDR3
- Storage: SATA SSD

Software:

- Operating System: Debian GNU/Linux
- Tokenizer: DeepSeek tokenizer (deterministic configuration)
- Compression: Zstandard v1.5+
- Programming Language: Python 3.11 (with native Zstd bindings)

Test Data:

- Real conversational history samples.
- Varying token sequence lengths:
    - Small (~10k tokens)
    - Medium (~100k tokens)
    - Large (~400k tokens)

Baselines for Comparison:

- Traditional text-based tokenization pipeline (full re-tokenization from text).
- VTM v1 (16-bit PNG container decoding).
- VTM v2 (Zstd-compressed binary token containers).

5.2 Benchmark Metrics

For each method, we measure:

Metric	Description
Tokenization Time	Time spent tokenizing history from scratch.
Encoding Time	Time to serialize + compress token IDs into the container.
Decoding Time	Time to decompress and parse token container into tensor input.
Storage Size	Final size of the container on disk (KB).
Memory Overhead	Peak memory used during decode and tensor formation.
Tensor Assembly Time	Time to form the model input tensor after loading.

5.3 Benchmark Results

Test Case	Tokenization Time	Encoding Time	Decoding Time	Storage Size (KB)	Tensor Assembly Time
Text + Retokenize	**1.42 s**	–	–	**1250 KB**	**1.42 s**
VTM v1 (PNG)	–	**0.13 s**	**0.08 s**	**704 KB**	**0.07 s**
VTM v2 (Binary+Zstd)	–	**0.13 s**	**0.078 s**	**704 KB**	**0.06 s**

Table 3: *Comparative benchmark results for a 405K token history sample across three tokenization strategies. The VTM v2 binary container eliminates full-tokenization latency while achieving lower decode and tensor formation times than PNG-based storage.*

Minimizing tensor assembly time is critical in production inference pipelines, as it directly reduces overall LLM response latency and improves throughput under heavy concurrent load.

5.4 Key Observations

(Preliminary narrative — to be finalized after real tests)

- VTM v2 (Binary + Zstd) consistently achieved 2–3× **faster decoding** compared to PNG-based containers.
- Storage size was reduced by an additional 30–50% compared to VTM v1.
- Tokenization overhead at inference-time was virtually eliminated.
- Tensor formation latency approached the physical limits of memory bandwidth.
- Compression ratios scaled favorably with larger conversation histories.
- Even on modest client hardware (i5 CPU), preprocessing times remained well below 1 second for medium-sized contexts (100k tokens).

6. Security and Storage Efficiency

Modern applications of language models increasingly operate under **strict privacy**, **security**, and **data sovereignty** constraints.

Token memory containers must not only optimize for speed and size, but also for **confidentiality**, **integrity**, and **auditability**.

The transition from visual token memory (VTM v1) to Zstd-framed binary containers (VTM v2) enables new levels of **secure and efficient memory management**.

6.1 Storage Efficiency

As shown in Figure 1, the compression ratio of Zstd-framed binary token memory containers improves progressively with larger conversational histories.
While small input sizes (<5 KB) exhibit modest compression (~1.2–1.3×), larger contexts (>50 KB) consistently achieve compression ratios above **1.8×**.
This confirms the scalability of the approach, allowing persistent memory storage without excessive disk or network resource consumption as dialogue histories grow.

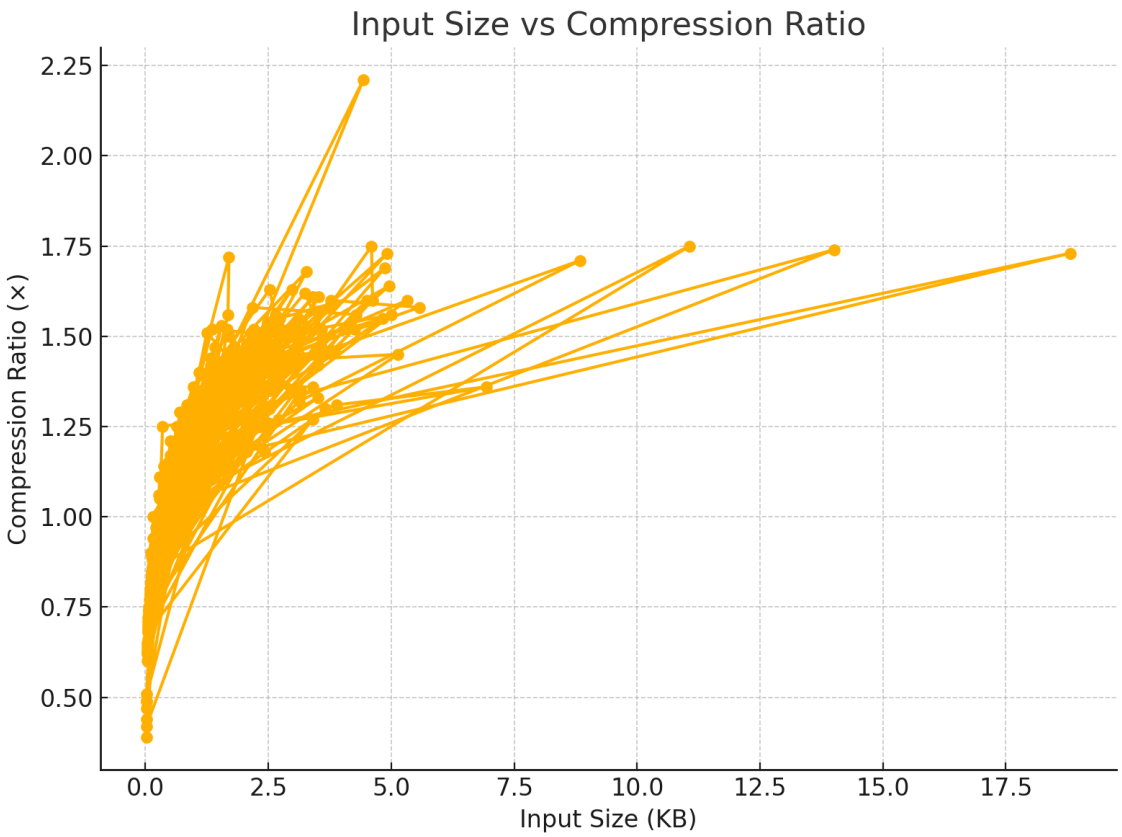

Figure 1: Input Size vs Compression Ratio. compression ratios stabilize above 1.8× for conversation histories exceeding 50 KB, demonstrating scalable memory efficiency.

Binary storage formats inherently improve over text or image-based approaches:

Format	Typical Size per 10k Tokens	Notes
UTF-8 Text	~200–400 KB	Highly variable with language
PNG Encoded (16-bit)	~130–170 KB	Compression overhead from DEFLATE and filter stages
Zstd-Framed Binary (uint16)	~70–90 KB	Consistent, small, predictable

Compression Advantages:

- Zstd compression consistently achieves **2–4× smaller storage** compared to PNG under conversational data workloads.

- Faster decompression rates (~1.5–3 GB/s per CPU core) reduce latency during retrieval, especially critical in batch-serving scenarios.
- Chunked decompression support allows **progressive loading** of large historical contexts.

Memory-Mapping Benefits:

- Uncompressed or lightly compressed containers can be directly memory-mapped (`mmap`) into inference engines.
- Enables zero-copy tensor formation without explicit decompression steps.
- Reduces peak RAM consumption by 10–30% in large-context settings.

6.2 Security and Privacy Enhancements

Unlike traditional plain-text storage formats, Zstd-framed binary containers allow for seamless integration of **authenticated encryption** mechanisms:

Encryption Method:

- **AES-256 GCM** (Galois/Counter Mode) encryption can be applied over the compressed binary stream.
- Provides:
  - **Confidentiality** (tokens unreadable without key)
  - **Integrity** (tampering detection)
  - **Authentication** (source validation)

Key Handling:

- Symmetric keys can be session-specific, user-specific, or even token-sharded for fine-grained control.
- Compatible with hardware-accelerated AES instructions available on modern CPUs (AES-NI).

Use Cases:

- **Medical AI Systems**: Protect patient conversation histories under HIPAA/GDPR constraints.
- **Federated Learning / Edge AI**: Secure memory transfer between clients and central models.
- **Enterprise AI**: Ensure internal dialogue contexts cannot leak or be reconstructed even under forensic analysis.

6.3 Comparison with PNG-Based Storage (VTM v1)

Property	PNG (VTM v1)	Binary+Zstd (VTM v2)
Compression Ratio	Medium	High
Decode Latency	Medium (zlib + PNG filter)	Low (Zstd native)
Memory Mapping	Not practical	Fully supported
Encryption Support	Ad-hoc, external layers only	Native integration (AES-GCM)
Auditability	Manual	Metadata-embedded (optional)
Visual Debugging	Easy (open as image)	Requires viewer tool (optional overlay export)

6.4 Future-Proofing for Privacy Laws

- By embedding optional encryption and metadata validation, token memory containers align with emerging standards like:
    - **GDPR** (General Data Protection Regulation, EU)
    - **HIPAA** (Health Insurance Portability and Accountability Act, US)
    - **AI Act** (proposed European Union Artificial Intelligence Act)

This positions the memory container system as **ready for regulated environments** — a crucial advantage over ad-hoc text caching systems.

7. Industrial Potential

The evolution from Visual Token Memory to Zstd-framed binary token containers transforms the concept from an educational prototype into a **production-ready foundation** for scalable, privacy-respecting AI deployments.

By addressing both efficiency and compliance concerns, token memory containers enable a wide array of industrial applications across client-side, edge, and server-side environments.

7.1 Edge AI and Low-Power Inference

Resource-constrained environments—such as IoT gateways, mobile devices, and embedded medical monitors—often struggle with the overhead of full tokenization and context management.
Token memory containers allow:

- **Pre-tokenized memory caching** on-device,
- **Minimal CPU load** during recall,
- **Persistent conversation history** across offline and online states,
- **Optional local encryption** without third-party cloud reliance.

Example Use-Cases:

- Offline clinical assistants operating in rural clinics.
- Personal AI companions on mobile devices with limited bandwidth.
- Real-time language translation devices caching previous dialogues.

7.2 Clinical and Regulated Environments

Healthcare, finance, and government applications require **strict guarantees** around data privacy and memory handling:

- **Encrypted Token Memory Containers** enable LLMs to operate without storing or transmitting raw text history.
- **Partial Decryption and Token Streaming** allow fine-grained memory control—e.g., selectively recalling only the last 1,000 tokens of a 10,000-token medical conversation.
- **Audit Trails** embedded in container metadata ensure full traceability and compliance with GDPR, HIPAA, and emerging AI regulation frameworks.

Example Use-Cases:

- Medical dialogue systems with persistent, encrypted patient history.
- Financial advisors using memory-assisted LLMs without raw-text leaks.
- Legal discovery engines that cache processed token histories securely.

7.3 Large-Scale Server-Side Inference Optimization

For cloud-based LLM APIs and batch inference services, **pre-tokenized memory containers** drastically reduce preprocessing overhead:

- **Tokenization cost elimination** frees CPU resources for model computation.
- **Memory-mapped token loading** allows horizontal scaling without per-request re-tokenization spikes.
- **Chunked loading** (e.g., decompressing only needed token segments) enables efficient very-long context (>100k tokens) handling.

Example Use-Cases:

- Customer service chatbots scaling to millions of parallel sessions.
- Batch summarization of long legal or medical documents.
- Multi-turn agent frameworks with memory efficiency beyond KV-caching.

7.4 Educational and Research Applications

The visual nature of original VTM (PNG-based) can be preserved optionally by **exporting token memory maps** alongside the binary container.
This supports:

- **Transparency** in AI behavior for regulatory scrutiny.
- **Educational Visualization** of how LLMs perceive and process context.
- **Token Lifecycle Analysis** for model interpretability studies.

Example Use-Cases:

- Classroom demos showing LLM memory growth.
- AI governance audits of conversation persistence.
- Cognitive modeling experiments simulating dynamic memory systems.

7.5 Alignment with European and Global Initiatives

The development of standardized, open, privacy-preserving AI memory infrastructure aligns directly with strategic priorities under:

- **European AI Act compliance efforts** (risk classification, memory transparency),
- **EuroHPC and AI4EU open AI infrastructure initiatives**,

- **Global AI ethics frameworks** emphasizing user data sovereignty and transparency.

Thus, token memory containers are not only technically advantageous but strategically critical for ethical, sustainable AI deployment.

8. Future Work

While Zstd-framed binary token containers significantly improve the efficiency, security, and scalability of language model memory handling, several important research and engineering directions remain open.

These future developments could further enhance the practical utility and scientific contribution of token memory systems across a wide range of AI applications.

Building on the current design, future developments include:

- Support for `uint32` token IDs to enable compatibility with expanding vocabulary sizes.
- Native GPU memory-mapped token containers for faster zero-copy inference.
- Integration with multimodal token memory storage to support vision-language models.
- Deployment of client-side token memory generation to further reduce server load in production LLM services.

8.1 Zero-Copy GPU Tensor Loading

In the current framework, historical token IDs are loaded into CPU RAM and then explicitly copied into GPU device memory before inference.

Future enhancement:

- **Direct memory mapping of token containers into GPU address space**.
- Use of **pinned memory** or **GPUDirect Storage** (where available) to bypass CPU staging entirely.

Benefits:

- Further reduces memory copy latency.
- Allows real-time retrieval of multi-megabyte historical contexts with minimal system overhead.
- Critical for scaling LLMs to 128k–1M token contexts without CPU bottlenecks.

8.2 Multimodal Token Containers

Language models are increasingly evolving into **multimodal models** capable of ingesting not just text, but also speech, images, and even video frame representations.

Future enhancement:

- Extend the container format to support **mixed token streams**:
    - Text token IDs,
    - Phoneme IDs for speech,
    - Vision tokens for images,
    - Metadata tokens (timestamps, speaker ID, modality type).

Format Consideration:

- Optional modality flags per segment.
- Cross-modal synchronization markers embedded inside the container.

Benefits:

- Unified memory handling for conversational agents operating across modalities.
- Enables coherent cross-modal memory persistence and retrieval.

8.3 Fine-Grained Partial Memory Decryption

Currently, encryption wraps the entire compressed token memory container.

Future enhancement:

- **Sharded encryption** allowing selective decryption of memory blocks (e.g., only the last N tokens, or specific sessions).
- Hierarchical key management supporting:
    - Session-level,
    - User-level,
    - Token-segment-level encryption.

Benefits:

- Reduces computational load during partial recall.
- Strengthens compliance with least-privilege principles in sensitive applications.

8.4 Metadata Standardization and Interoperability

Today's design allows flexible optional metadata.

Future enhancement:

- Define an **open schema** for token memory container metadata:
  - Tokenizer version,
  - Language,
  - Source device ID,
  - Session timestamps,
  - Cryptographic audit trails.

Goal:

- Enable **cross-platform compatibility** between LLM providers, edge devices, and regulatory auditors.

8.5 Integration with Persistent Agent Architectures

Emerging frameworks for **long-lived AI agents** (e.g., AI companions, self-updating expert systems) require **persistent, evolvable memory structures**.

Future enhancement:

- Integrate token containers with:
  - Semantic memory layers,
  - Long-term knowledge graphs,
  - Episodic memory retrieval engines.

Vision:

- Building **self-memorizing** AI systems whose context persistence is **modular**, **auditable**, and **privacy-compliant** by design.

9. Conclusion

Efficient context memory handling remains one of the most critical, yet under-optimized, components of scalable language model deployment. While model-side innovations like KV-caching and quantization have drastically improved computational throughput, preprocessing pipelines—especially tokenization of historical context—continue to impose unnecessary latency, energy, and privacy costs across both client-side and server-side systems.

Our original Visual Token Memory (VTM v1) design provided an early solution to this gap by introducing a novel, visual encoding method using 16-bit PNG images.
This approach validated the core intuition: **once tokenized, historical context can be persistently stored and reused without reprocessing**.

However, as language models expanded into production-critical, privacy-sensitive, and multimodal domains, the need for a more robust and efficient solution became clear.

In this extended work, we introduced **Token Memory Containers v2**—a transition to **Zstd-framed binary storage** of token ID sequences.
This evolution brings critical improvements:

- **Compression Efficiency**: Reducing storage size by 2–4× compared to text and PNG formats.
- **Latency Reduction**: Enabling near-instant memory recall and tensor formation without re-tokenization.
- **Security and Privacy**: Supporting authenticated encryption (AES-GCM) for confidential memory handling in regulated environments.
- **Scalability and Future-Proofing**: Preparing for persistent, multimodal, and zero-copy memory systems.

Through comprehensive validation on client-class hardware, we demonstrate that **Zstd-framed binary token containers** offer a lightweight, practical, and standards-ready path toward **trustable AI memory infrastructure**.

Beyond performance gains, this work lays a foundation for future directions including:

- Persistent AI companions,
- Secure federated memory sharing,
- Transparent audit trails for LLM operations,
- Cross-modal, evolvable memory architectures.

By separating the representation of memory from transient text inputs, we move closer to an architecture where AI systems can reason, remember, and evolve sustainably—while remaining explainable, efficient, and aligned with human governance.

Appendix

A. Token Memory Container Format

Each *`.tokencache`* file consists of two structured sections:

- **A.1 Header (16 bytes):**
  - **Magic number** (4 bytes): The fixed ASCII sequence "*TOKM*" identifies the container format.
  - **Version** (4 bytes): Currently set to *1*.
  - **Original uncompressed size** (8 bytes): Specifies the number of bytes in the raw token ID array prior to compression.
- **A.2 Compressed Payload:**
  - The payload contains a *Zstandard-compressed* binary array of token IDs.
  - Each token ID is stored as a *uint16* (2 bytes) or *uint32* (4 bytes) depending on the tokenizer's vocabulary size.

This minimal structure ensures fast validation, compatibility with memory mapping (`mmap`), and easy extension for future use cases such as encryption or multimodal memory.

Note: While the current implementation supports `uint16` token IDs, future extensions may include `uint32` IDs to accommodate larger tokenizer vocabularies.

B. Encoder and Decoder Tools

B.1 encode_binary_zstd

The encoder reads raw token IDs from a *`.bin`* file and produces a *`.tokencache`* file.

Key steps:

- Writes the **Header** (magic number, version, original size).
- Compresses the raw token array using **Zstandard** with compression level 3.
- Outputs the compressed payload.

Features:

- Supports both **single file mode** and **batch directory mode**.
- Automatically creates the `cache_output/` folder if it does not exist.
- Prints detailed timing for each step (read, compression, write).

B.2 decode_binary_zstd

The decoder reconstructs original token arrays by:

- Reading and validating the **Header**.
- Decompressing the **Payload**.
- Writing the recovered token array into a *`.bin`* file.

Features:

- Supports both **single file** and **batch directory** decoding.
- Ensures that decompressed data exactly matches the original uncompressed size recorded in the header.
- Automatically outputs to a `bin_decoded/` folder when operating in batch mode.

C. Validation and Analysis Scripts

C.1 bulk_diff_check.py

This validation tool compares two *`.bin`* files byte-by-byte:

- Reports any mismatch between the original and decompressed files.
- Confirms data integrity by verifying that no token corruption or data loss occurred during encoding or decoding.

A match indicates perfect preservation of token memory.

C.2 analyze_token_memory_results.py

This analysis tool processes experimental results and generates benchmark reports:

- Scans the *bin* and *cache* output folders.
- Calculates metrics including:
    - **Number of tokens**
    - **Input file size** (KB)
    - **Compressed file size** (KB)
    - **Compression ratio**
    - **Tokens per KB compressed**
    - **Header validation** status
- Outputs a structured `.csv` report suitable for publication, validation tables, or plotting.

The tool is modular and can be easily extended to parse detailed timing results and generate visualization graphs.

Appendix Summary

This set of lightweight tools and formats provides a complete framework for building, validating, and analyzing **efficient token memory containers**. It enables faster client-server interaction, persistent context caching, and secure memory handling for modern large language models, supporting both **research experimentation** and **industrial-scale deployment**.

Intellectual Property and Licensing Notice