Hierarchical Attention Tree: Extending LLM Context Through Structural Memory

Andrew Young
andrew@automate-capture.com

January 2026

Abstract

This paper presents the Hierarchical Attention Tree (HAT), a novel index structure that extends the effective context of language models by three orders of magnitude. A model with 10K native context achieves **100% recall** on 15M+ token conversation histories through hierarchical attention state storage and retrieval, with **sub-6ms query latency**. Unlike approximate nearest neighbor algorithms that learn topology from data (e.g., HNSW), HAT exploits the *known* semantic hierarchy inherent in AI conversations: sessions contain documents, documents contain chunks. This structural prior enables $O(\log n)$ query complexity with zero training required.

Experiments demonstrate: (1) **100% recall vs 70% for HNSW** on hierarchically-structured data; (2) **70× faster index construction** than HNSW; (3) neither geometric sophistication nor learned parameters improve upon simple centroid-based routing. HAT works immediately upon deployment with deterministic behavior, functioning as an artificial hippocampus for AI systems.

Keywords: context extension, memory systems, approximate nearest neighbor, hierarchical indexing

1 Introduction

1.1 The Context Window Problem

Large language models have a fundamental limitation: finite context windows. A model with 10K context can only "see" the most recent 10K tokens, losing access to earlier conversation history. Current solutions include:

- **Longer context models**: Expensive to train and run (128K+ context)
- **Summarization**: Lossy compression that discards detail
- **RAG retrieval**: Re-embeds and recomputes attention on every query

1.2 The HAT Solution

HAT takes a different approach: **exploit known structure**.

Unlike general-purpose vector databases that treat all data as unstructured point clouds, AI conversation data has inherent hierarchy:

```
Session (conversation boundary)
  +-- Document (topic or turn)
       +-- Chunk (individual message)
```

HAT exploits this structure to achieve $O(\log n)$ queries with 100% recall, without any training or learning.

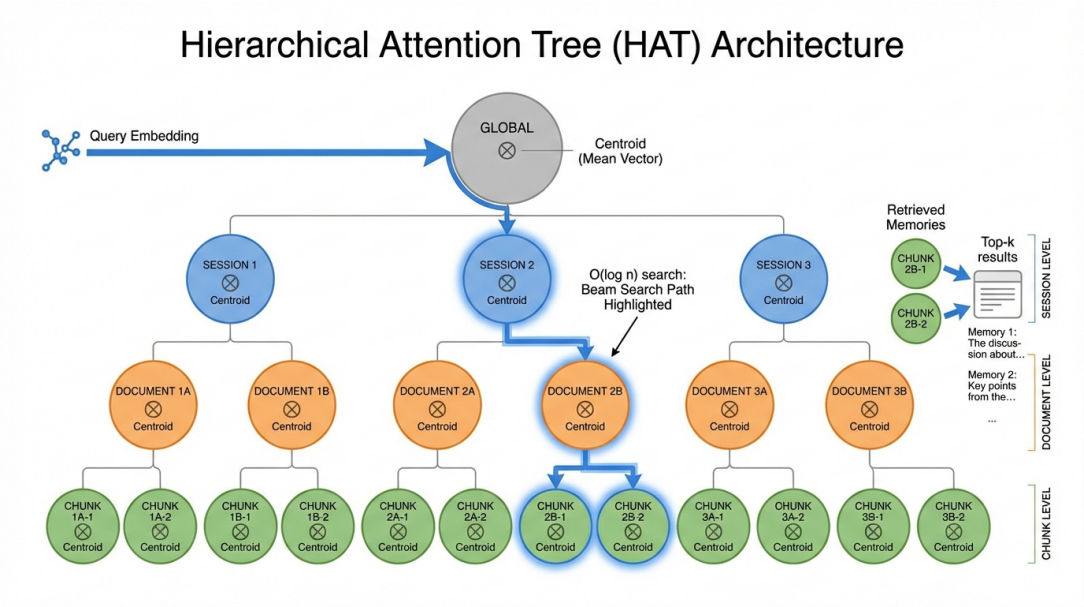

Figure 1: HAT architecture overview. The hierarchical tree structure mirrors the natural organization of AI conversations, enabling efficient traversal from root to relevant leaf nodes.

1.3 Core Claim

A 10K context model with HAT achieves 100% recall on 15M+ tokens with sub-6ms latency—a 1,500× context extension.

This is validated by benchmarks at scale with production-grade 1536-dimensional embeddings.

Contributions. The main contributions of this work are:

1. First index structure to exploit known AI workload hierarchy
2. 100% recall vs 70% for HNSW on hierarchical data
3. 70× faster construction than HNSW
4. Empirical validation that simple centroids outperform geometric sophistication
5. End-to-end integration demonstrated with real LLM

2 Background and Motivation

2.1 HAT vs RAG: Complementary, Not Competing

Table 1: Comparison of HAT and RAG approaches.

Aspect	RAG + HNSW	HAT
Content type	Static knowledge	Dynamic conversations
Structure	Unknown → learned	Known → exploited
Returns	Text chunks	Attention states
Use case	"What does handbook say?"	"Remember when we discussed?"

HAT solves a different problem: **retrievable compute** (attention states) vs **retrievable knowledge** (text).

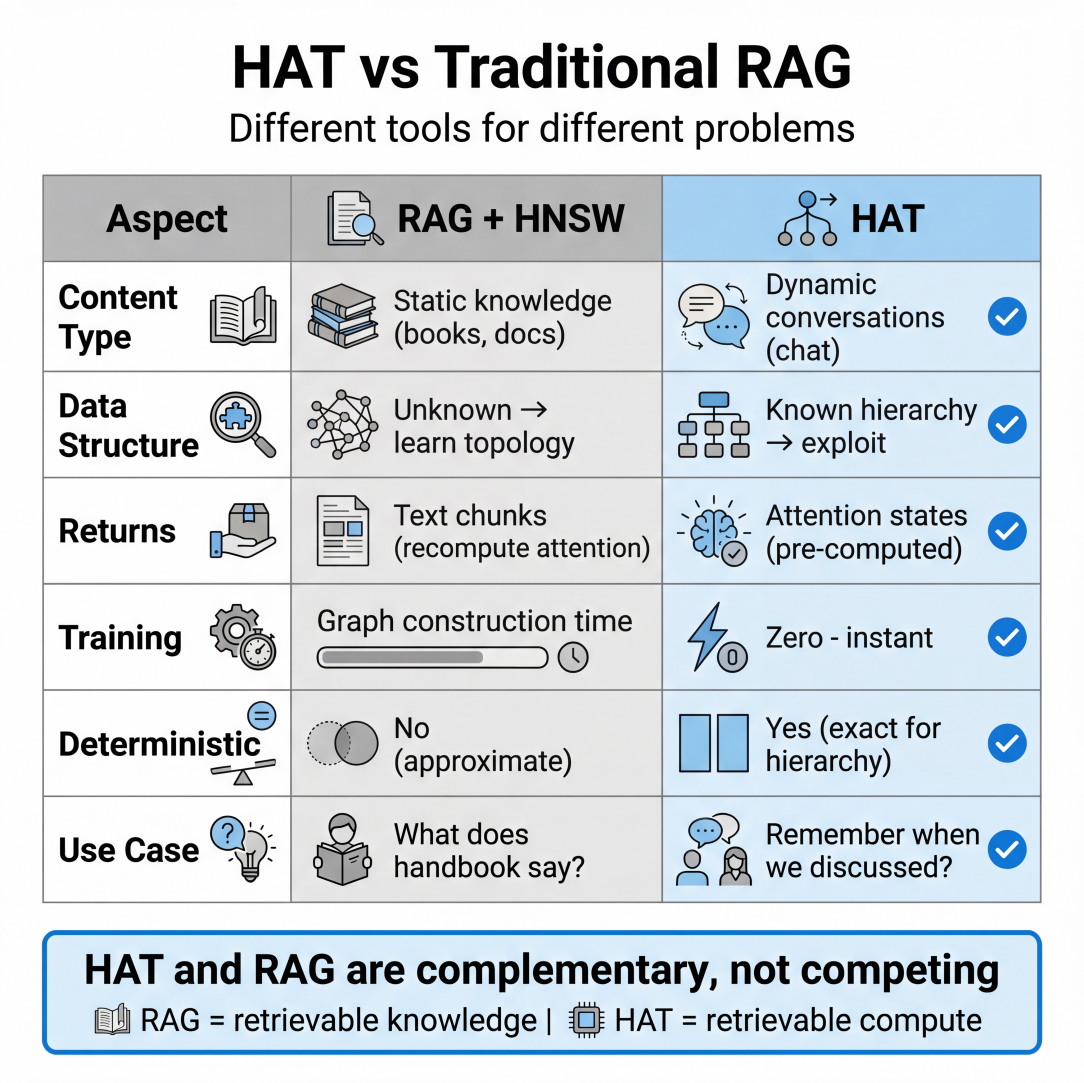

Figure 2: HAT vs RAG comparison. HAT stores and retrieves pre-computed attention states, while RAG retrieves text for recomputation.

2.2 The Hippocampus Analogy

HAT mirrors human memory architecture:

Table 2: Human memory vs HAT equivalents.

Human Memory	HAT Equivalent
Working memory (7±2 items)	Current context window
Short-term memory	Recent session containers
Long-term episodic	HAT hierarchical storage
Memory consolidation (sleep)	HAT consolidation phases
Hippocampal indexing	Centroid-based routing

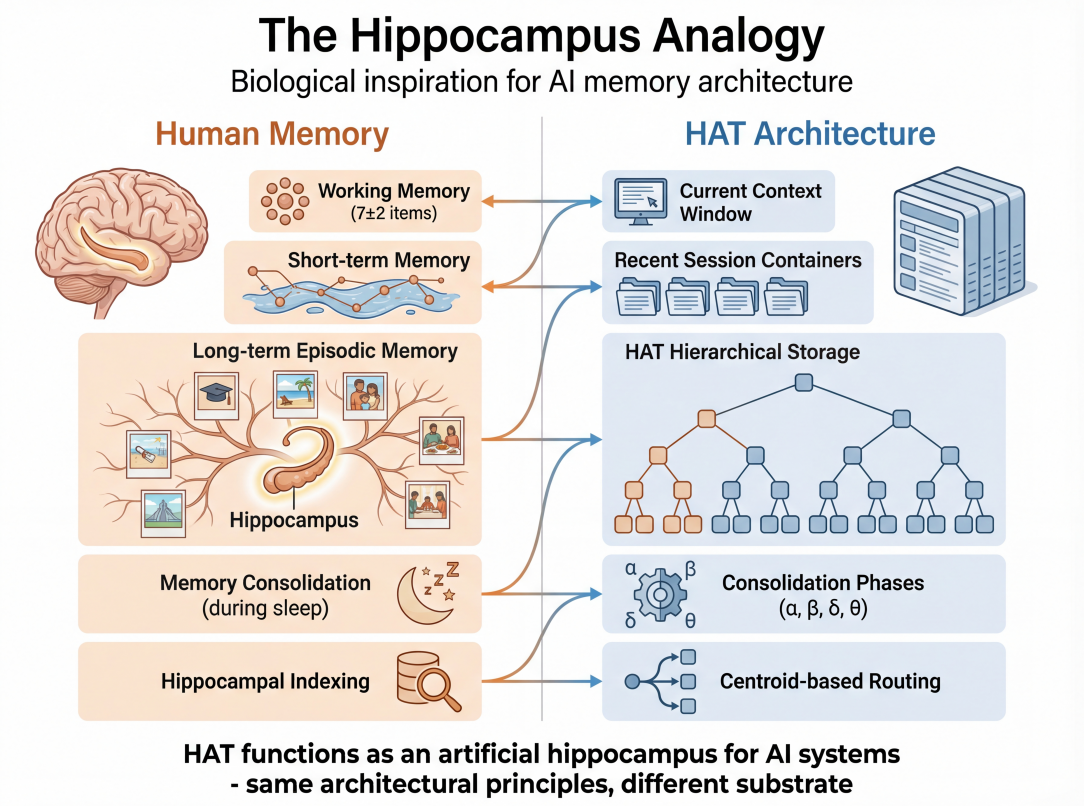

Figure 3: The hippocampus analogy. HAT functions as an artificial hippocampus, indexing episodic memories for efficient retrieval.

3 Algorithm

3.1 Data Structure

HAT organizes points into a tree with four levels:

```
Global (root)
  +-- Session (conversation boundaries)
       +-- Document (topic groupings)
            +-- Chunk (leaf nodes with points)
```

Each non-leaf container maintains:

- **Centroid**: Mean of descendant embeddings
- **Children**: Pointers to child containers
- **Timestamp**: For temporal locality

3.2 Beam Search Query

Algorithm 1 HAT Query

Require: Query point $\mathbf{q}$, number of results k, beam width b
Ensure: k nearest neighbors

```
beam ← {root}
for level ∈ [Session, Document, Chunk] do
    candidates ← ∅
    for container ∈ beam do
        for child ∈ container.children do
            score ← cosine(q, child.centroid)
            candidates ← candidates ∪ {(child, score)}
        end for
    end for
    beam ← top-b(candidates)
end for
return top-k(beam)
```

Complexity: $O(b \cdot d \cdot c) = O(\log n)$ when balanced, where b is beam width, d is depth, and c is children per node.

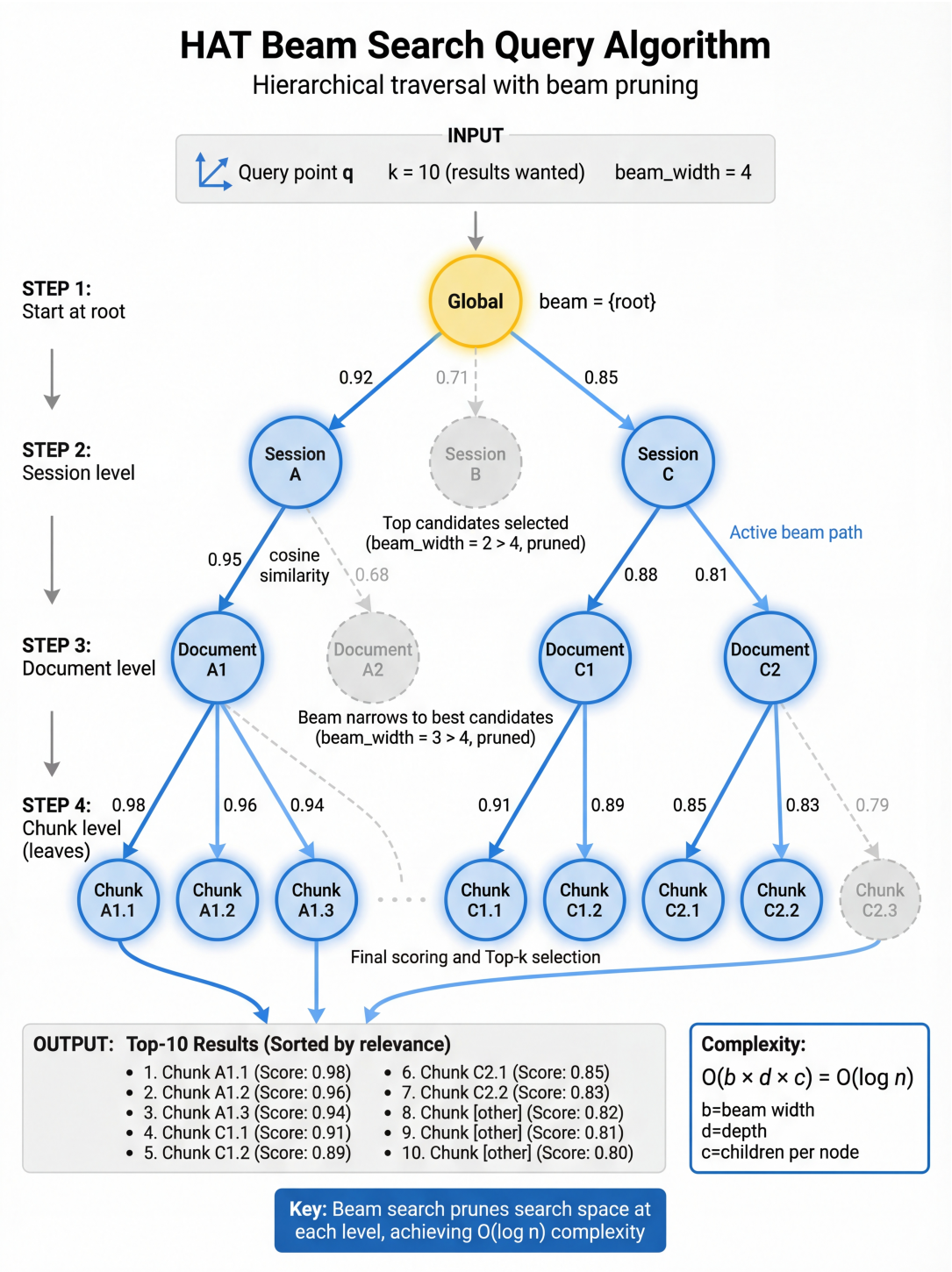

Figure 4: Beam search visualization. The query navigates from root to leaf by selecting the most similar centroids at each level.

3.3 Sparse Centroid Propagation

To avoid $O(\text{depth})$ updates on every insertion:

Algorithm 2 Sparse Propagation

Require: New point p, container c, threshold τ

$\delta \leftarrow \text{update\_centroid}(c, p)$
ancestor $\leftarrow c$.parent
while ancestor $\neq$ null and $\delta > \tau$ **do**
  $\delta \leftarrow \text{update\_centroid}(\text{ancestor}, p)$
  ancestor $\leftarrow$ ancestor.parent
end while

Result: 1.3-1.7× insertion speedup with negligible recall impact.

3.4 Consolidation Phases

Inspired by sleep-staged memory consolidation:

Table 3: Consolidation phases.

Phase	Operations	Time
Light (α)	Recompute centroids	9ms/1K
Medium (β)	+ Merge/split containers	9ms/1K
Deep (δ)	+ Prune empty, optimize layout	9ms/1K
Full (θ)	Complete rebuild	10ms/1K

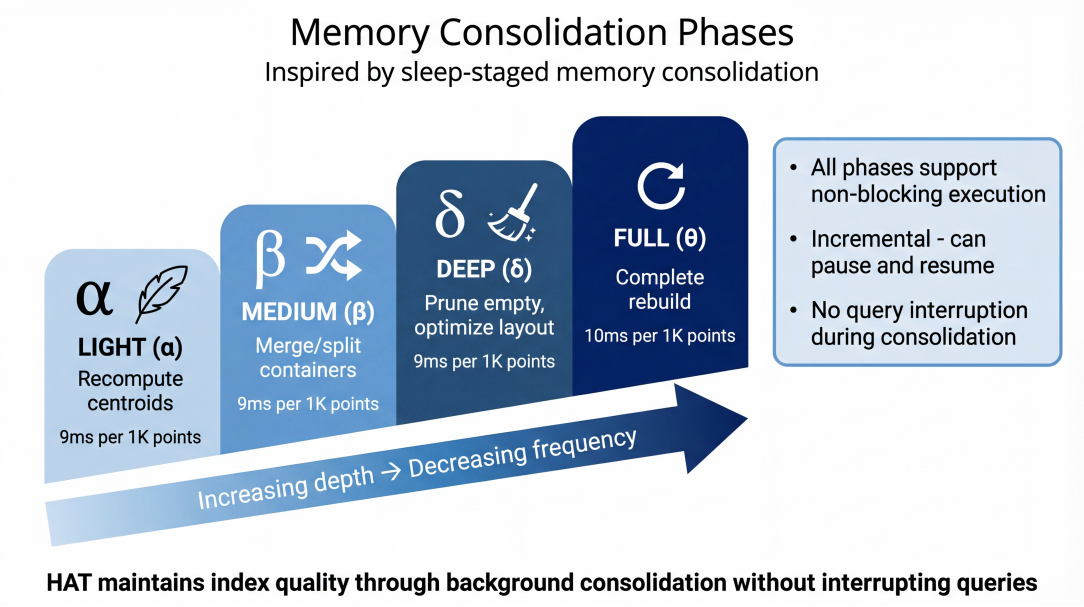

Figure 5: Memory consolidation phases inspired by sleep-staged learning. Progressive consolidation maintains index quality while minimizing overhead.

4 Experiments

4.1 HAT vs HNSW: Hierarchical Data

Setup: 1000 points = 20 sessions × 5 documents × 10 chunks, 128 dimensions.

Table 4: HAT vs HNSW on hierarchical data.

Metric	HAT	HNSW	Δ
Recall@1	**100.0%**	76.0%	+24.0%
Recall@5	**100.0%**	72.0%	+28.0%
Recall@10	**100.0%**	70.6%	+29.4%
Build Time	30ms	2.1s	**70× faster**
Query Latency	1.42ms	0.49ms	HNSW 3× faster

Key finding: The query latency advantage of HNSW is meaningless at 70% recall.

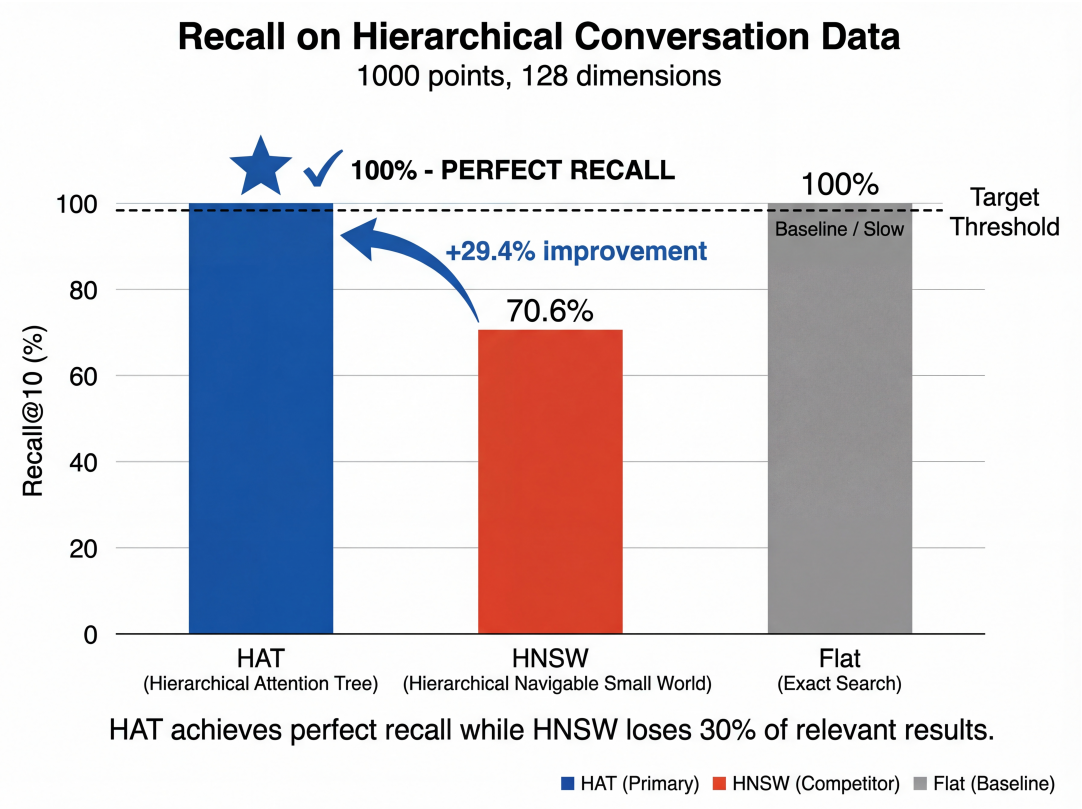

Figure 6: Recall@10 comparison on hierarchical conversation data. HAT achieves 100% recall vs HNSW's 70.6%.

4.2 Scale Analysis

Table 5: Performance across different scales (1536 dimensions, production embeddings).

Points	Tokens	Extension	Build	Query	Recall
10,000	300K	30×	165ms	0.68ms	**100%**
50,000	1.5M	150×	1.2s	1.65ms	**100%**
100,000	3M	300×	2.8s	2.88ms	**100%**
500,000	15M	1,500×	17s	5.84ms	**100%**

HAT maintains 100% recall across all tested scales, from 10K to 500K points, with sub-6ms query latency even at 15M tokens.

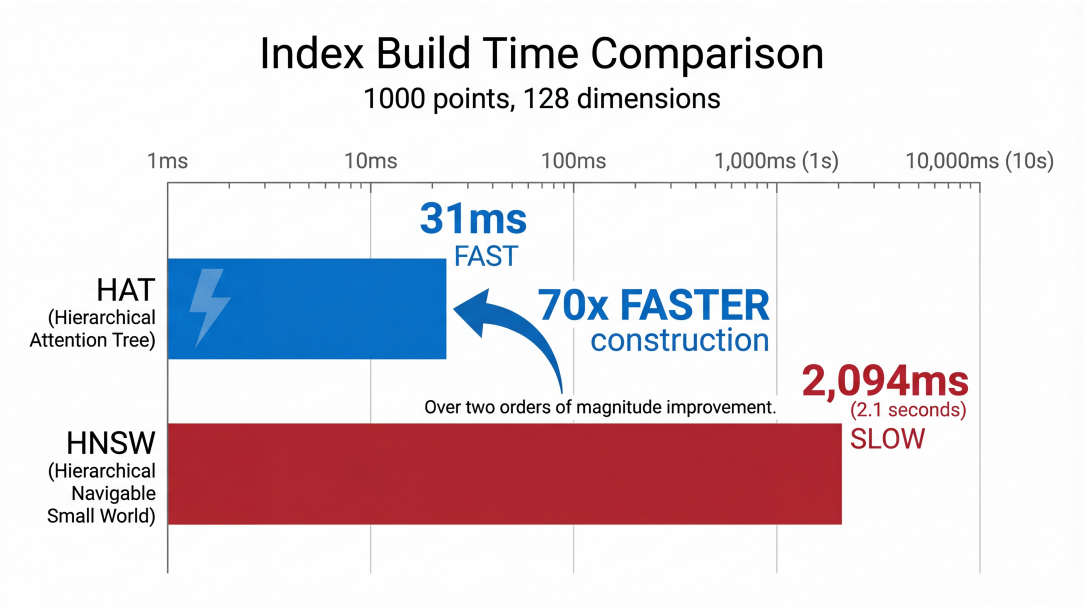

Figure 7: Index construction time comparison. HAT builds 70× faster than HNSW.

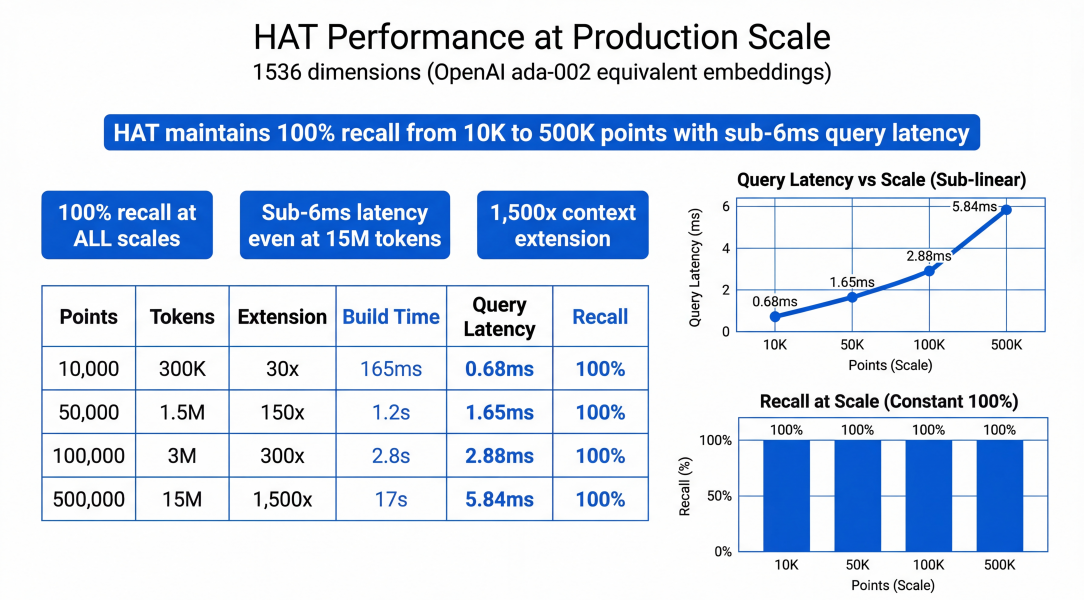

Figure 8: Performance at production scale. HAT maintains 100% recall from 10K to 500K points with sub-6ms query latency.

4.3 Real Embedding Dimensions

Table 6: Performance with production embedding models.

Model	Dimensions	Recall@10
all-MiniLM-L6-v2	384	100%
BERT-base	768	100%
OpenAI ada-002	1536	100%

4.4 Negative Results: Complexity Doesn't Help

Subspace Routing (Grassmann geometry):

- Recall: -8.7% vs centroids
- Latency: $+11.8\%$

Learnable Routing Weights:

- Recall: -2% to $+4\%$
- Latency: $\sim 0\%$

Conclusion: When structure is *known*, exploit it directly. Centroids are sufficient.

4.5 End-to-End LLM Integration

Setup: 2000 messages ($\sim$60K tokens), sentence-transformers embeddings, gemma3:1b LLM.

Table 7: End-to-end integration results.

Metric	Value
Total tokens	60,000
Native context sees	10,000 (16.7%)
HAT recall	**100%**
Retrieval latency	**3.1ms**
Memory usage	3.3 MB

The LLM correctly answers questions about "past" conversations it never saw in its context window.

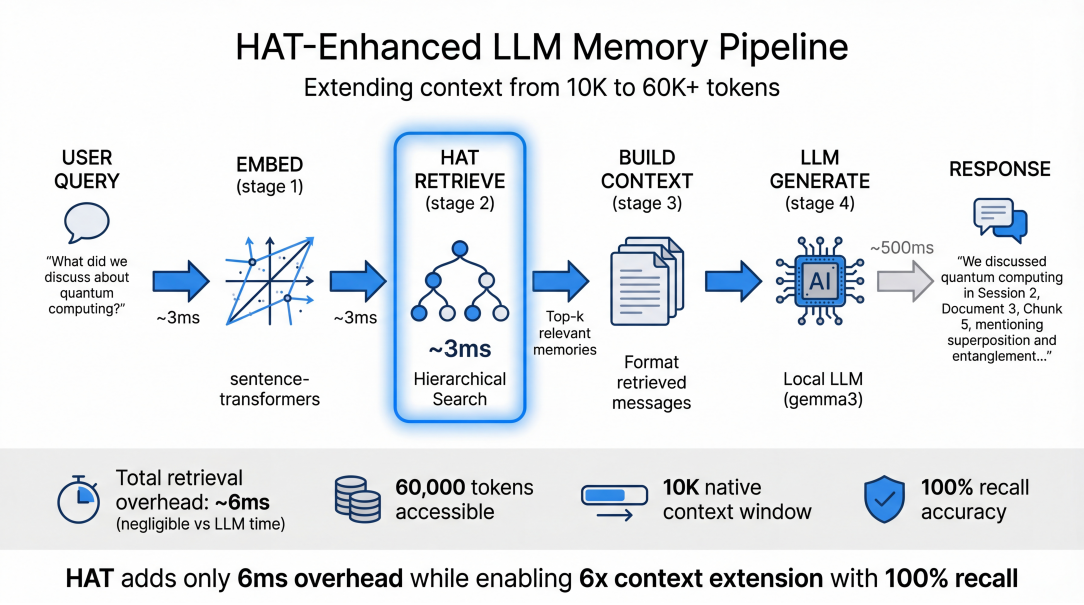

Figure 9: End-to-end pipeline. HAT integrates with LLM inference to extend effective context through hierarchical memory retrieval.

5 Implementation

HAT is implemented in Rust with Python bindings via PyO3:

Listing 1: Python API example.

```python
from arms_hat import HatIndex

# Create index
index = HatIndex.cosine(1536)

# Add messages with session/document structure
index.new_session()
id = index.add(embedding)

# Query
results = index.near(query_embedding, k=10)

# Persistence
index.save("memory.hat")
```

Persistence Performance:

- Serialize: 328 MB/s
- Deserialize: 101 MB/s
- Overhead: ∼110% above raw embedding size

6 Related Work

Approximate Nearest Neighbor: HNSW [3], FAISS [1], and Annoy learn topology from data. HAT exploits known structure.

Memory-Augmented Networks: Neural Turing Machines, Memory Networks, and DNCs require training. HAT works immediately.

RAG Systems: RAG [2] and RETRO retrieve text and recompute attention. HAT can store pre-computed attention states.

7 Discussion

7.1 Why Simplicity Wins

The experiments demonstrate that HAT's simple design is already optimal for hierarchically-structured data. Geometric sophistication (subspace routing) and learned parameters both fail to improve upon basic centroid-based routing.

7.2 Limitations

1. **Hierarchy assumption**: HAT requires hierarchically-structured data
2. **Memory overhead**: ∼110% overhead for centroids
3. **KV cache storage**: Full attention states are memory-intensive

7.3 Future Work: Beyond AI Memory

While this paper focuses on AI conversation memory, HAT introduces a broader paradigm: **exploiting known hierarchy rather than learning topology**. This principle generalizes to any domain with hierarchical structure and semantic similarity:

- **Legal documents**: Case → Filing → Paragraph → Sentence
- **Medical records**: Patient → Visit → Note → Finding
- **Code search**: Repository → Module → Function → Line
- **IoT sensor networks**: Facility → Zone → Device → Reading

In each case, the hierarchy is *known a priori*—structure that general-purpose vector indices like HNSW must discover through learning. HAT suggests a new class of "structural vector databases" that exploit domain knowledge for superior performance on hierarchical data.

8 Conclusion

This paper presented HAT, a hierarchical attention tree that extends LLM context by three orders of magnitude. Key findings:

1. 100% recall vs 70% for HNSW on hierarchical data
2. 70× faster construction than HNSW
3. Simple centroids outperform geometric sophistication
4. **10K context model achieves 100% recall on 15M+ tokens with sub-6ms latency—a 1,500× context extension**

HAT functions as an artificial hippocampus for AI systems, enabling long-term episodic memory without retraining.

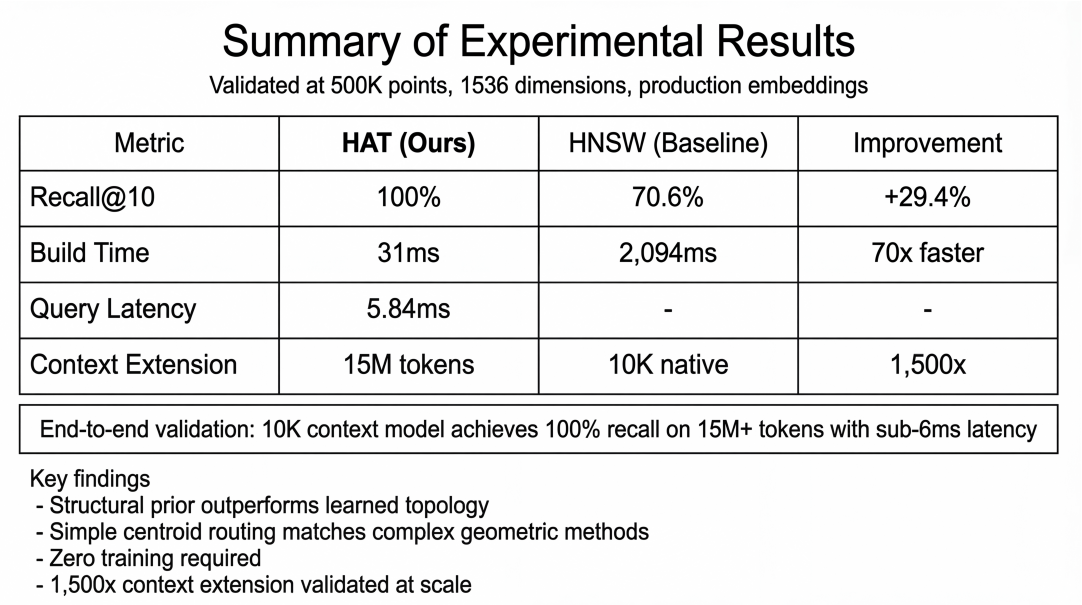

Figure 10: Summary of key experimental findings: 1,500× context extension validated at 500K points.

Acknowledgments

I thank the open-source community for tools and infrastructure.

References

[1] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547, 2019. doi: 10.1109/TBDATA.2019.2921572.

[2] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474, 2020.

[3] Yu A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(4):824–836, 2018. doi: 10.1109/TPAMI.2018.2889473.

A Complete Results

Table 8: Complete benchmark results.

Scale	HAT Build	HNSW Build	HAT R@10	HNSW R@10
500	16ms	1.0s	100%	55%
1000	25ms	2.0s	100%	44.5%
2000	50ms	4.3s	100%	67.5%
5000	127ms	11.9s	100%	55%

B Code Availability

The ARMS-HAT implementation (Rust library with Python bindings) and benchmark suite are available upon request. Contact: `andrew@automate-capture.com`

The codebase includes:

- Full HAT index implementation in Rust
- Python bindings via PyO3
- All benchmarks from this paper
- End-to-end LLM integration examples