ARMS: A Spatial Memory Fabric for AI Systems

Position IS Relationship

Andrew Young
andrew@automate-capture.com

January 2026

Abstract

This paper introduces ARMS (Attention Reasoning Memory Store), a spatial memory fabric that enables AI systems to store and retrieve computed states by their native dimensional coordinates. Unlike traditional databases that require explicit relationships through foreign keys or learned topology through approximate nearest neighbor algorithms, ARMS operates on a fundamental principle: **position IS relationship**. Proximity in high-dimensional space defines semantic connection without explicit declaration.

ARMS reduces memory operations to five primitives: **Point** (any-dimensional vectors), **Proximity** (relationship measurement), **Merge** (composition), **Place** (existence in space), and **Near** (retrieval by similarity). This minimal abstraction enables a hexagonal architecture where storage backends, index algorithms, and APIs can be swapped without changing core logic.

The framework provides the foundation for specialized index adapters like HAT (Hierarchical Attention Tree), demonstrating that domain-specific structure can be exploited for superior performance. ARMS functions as an artificial hippocampus—enabling AI systems to form, consolidate, and retrieve episodic memories through spatial organization rather than explicit indexing.

Keywords: spatial memory, AI memory systems, vector databases, hexagonal architecture, episodic memory

1 Introduction

1.1 The Memory Problem in AI

Large language models and AI agents face a fundamental limitation: they lack persistent, retrievable memory beyond their context window. Current approaches include:

- **Extended context**: Expensive, doesn't scale beyond training length
- **RAG retrieval**: Retrieves text, requires recomputation of attention
- **Vector databases**: Treat all data as unstructured point clouds
- **External memory**: Key-value stores with explicit indexing

None of these approaches preserve the *native representation* of computed states. When an LLM processes text, it produces attention states in high-dimensional space. Current systems project, compress, or discard these states rather than storing them directly.

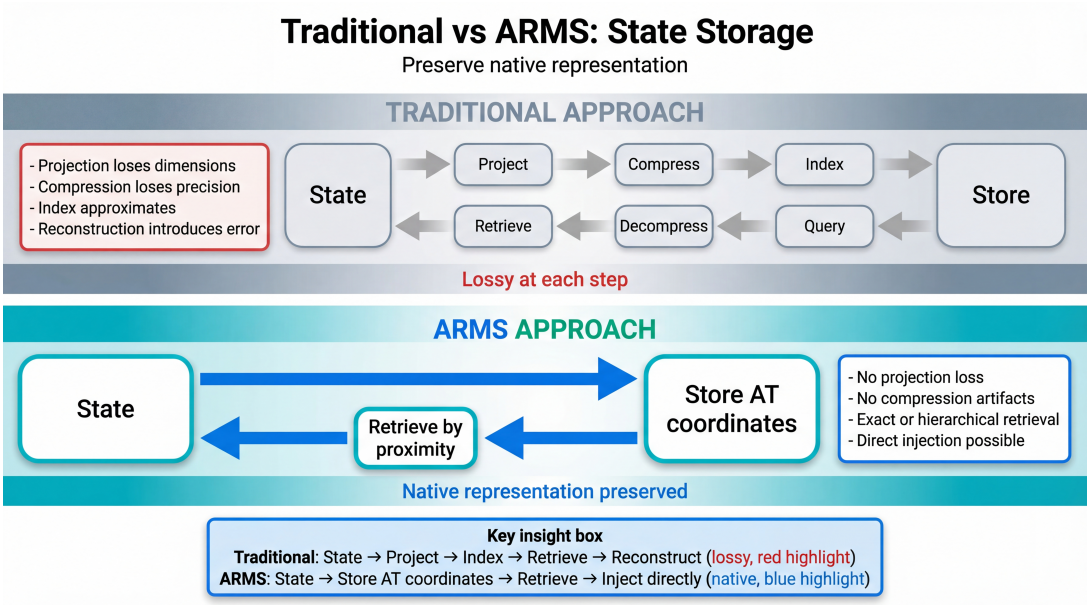

Figure 1: Traditional approaches vs ARMS: Traditional systems project, compress, and approximate states at each step, introducing cumulative error. ARMS stores states at their native coordinates and retrieves by proximity, preserving the original representation.

1.2 The ARMS Insight

ARMS takes a different approach:

> **Store states at their native coordinates. Retrieve by proximity. Position IS relationship.**

This insight has three implications:

1. **No projection loss**: States are stored in their original dimensionality
2. **No explicit relationships**: Semantic similarity is spatial proximity
3. **No learned topology**: Structure can be known or exploited, not discovered

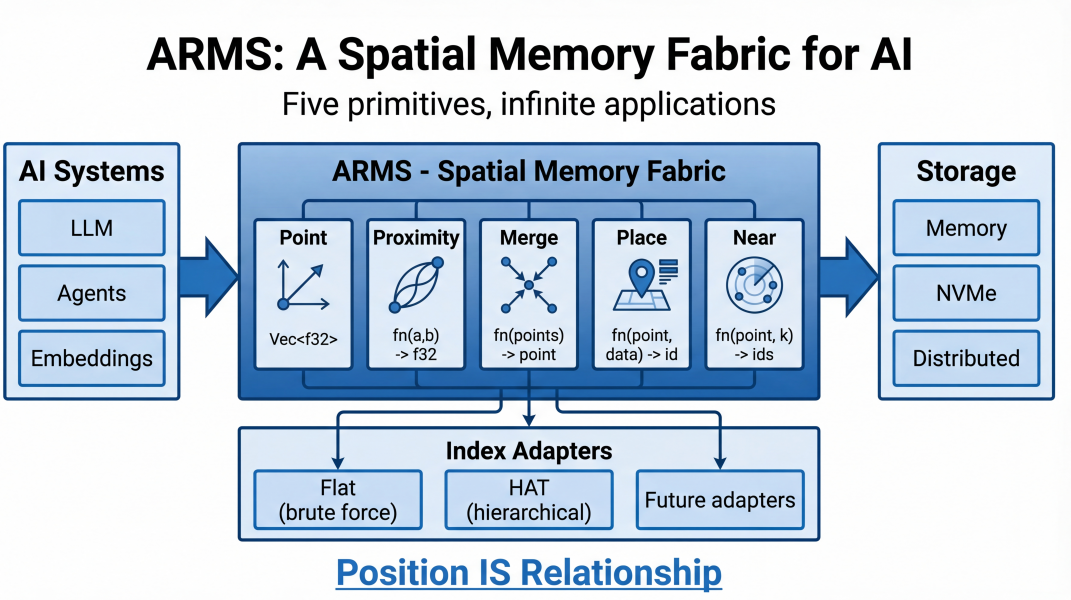

Figure 2: ARMS architecture overview. The five primitives (Point, Proximity, Merge, Place, Near) form the core, with swappable storage and index adapters.

1.3 Contributions

This paper makes the following contributions:

1. A **five-primitive abstraction** for spatial memory (Point, Proximity, Merge, Place, Near)
2. A **hexagonal architecture** enabling swappable storage, index, and API adapters
3. The **"position is relationship"** principle for AI memory systems
4. A **foundation framework** demonstrated through the HAT index adapter

2 The Five Primitives

ARMS reduces all memory operations to five primitives. This minimal surface area enables maximum flexibility while maintaining semantic clarity.

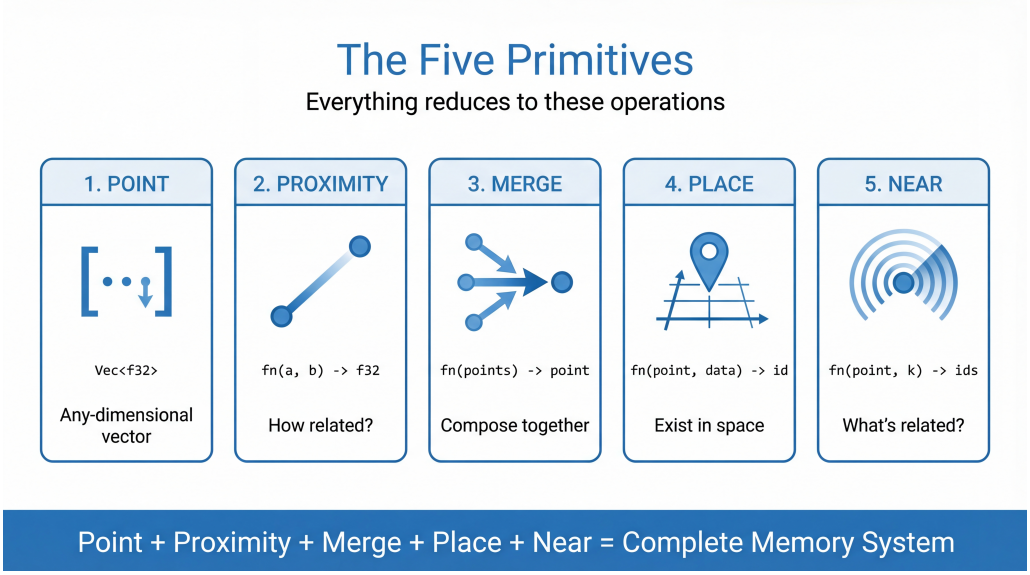

Figure 3: The five primitives of ARMS: Point (representation), Proximity (relationship), Merge (composition), Place (storage), and Near (retrieval). These operations form the complete interface for spatial memory.

Table 1: The five primitives of ARMS.

Primitive	**Signature**	**Purpose**
Point	`Vec<f32>`	Any-dimensional vector representation
Proximity	`fn(a, b) -> f32`	Measure how related two points are
Merge	`fn(points) -> point`	Compose multiple points into one
Place	`fn(point, data) -> id`	Store a point in the space
Near	`fn(point, k) -> ids`	Find k most related points

2.1 Point: The Universal Representation

A Point is simply a vector of floating-point numbers:

Listing 1: Point definition.

```
pub struct Point {
    dims: Vec<f32>,
}

impl Point {
    pub fn new(dims: Vec<f32>) -> Self;
    pub fn dimensionality(&self) -> usize;
    pub fn magnitude(&self) -> f32;
    pub fn normalize(&self) -> Point;
}
```

Points are dimensionality-agnostic. The same ARMS instance can store 768-dimensional BERT embeddings or 1536-dimensional OpenAI embeddings—the primitives don't change.

2.2 Proximity: Relationship Without Declaration

Proximity functions measure how related two points are:

Table 2: Built-in proximity functions.

Function	Range	Use Case
Cosine	$[-1, 1]$	Semantic similarity (direction matters)
Euclidean	$[0, \infty)$	Spatial distance (magnitude matters)
DotProduct	$(-\infty, \infty)$	Raw correlation
Manhattan	$[0, \infty)$	L1 distance

The key insight: **proximity replaces foreign keys**. In a relational database, you declare relationships explicitly. In ARMS, relationships emerge from spatial position.

2.3 Merge: Composition Without Loss

Merge combines multiple points into a single representative:

- **Mean**: Arithmetic average (default)
- **WeightedMean**: Importance-weighted average
- **MaxPool**: Element-wise maximum

Merge enables hierarchical summarization. A conversation can be represented by the merge of its messages; a session by the merge of its conversations.

2.4 Place and Near: The Memory Interface

Place stores a point with associated data:

Listing 2: Place and Near operations.

```
// Store
let id = arms.place(embedding, blob)?;

// Retrieve
let neighbors = arms.near(&query, k)?;
```

This is the complete memory interface. Everything else—storage backends, index algorithms, APIs—is implementation detail.

3 Hexagonal Architecture

ARMS follows hexagonal (ports-and-adapters) architecture. The core domain contains pure math with no I/O. Ports define trait contracts. Adapters provide swappable implementations.

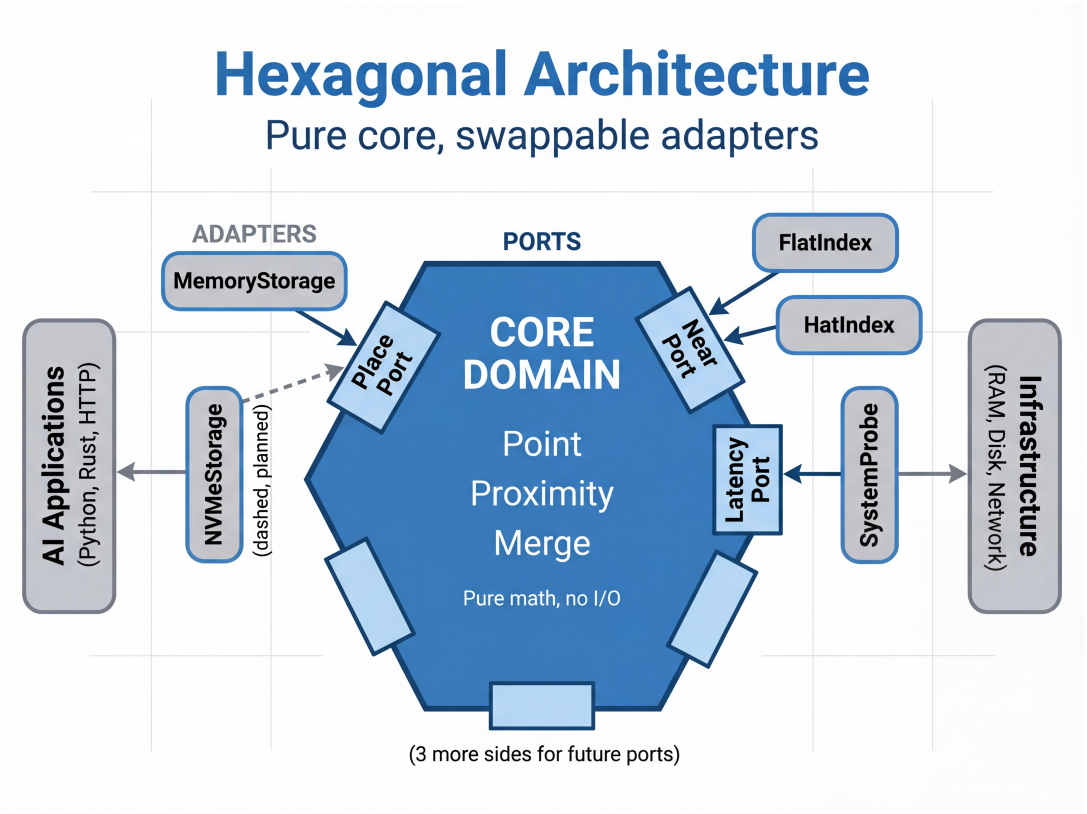

Figure 4: Hexagonal architecture of ARMS. The core domain contains pure math with no I/O. Ports define trait contracts. Adapters provide swappable implementations for storage, indexing, and APIs.

3.1 Core Domain

The core contains:

- **Point**: Vector representation
- **Id**: Unique identifiers
- **Blob**: Associated data
- **Proximity**: Relationship measurement
- **Merge**: Point composition

No I/O, no side effects, pure functions. This enables testing without mocks and reasoning without context.

3.2 Ports

Ports define what the system needs without specifying how:

Listing 3: Port definitions.

```
pub trait Place {
    fn place(&mut self, point: Point, blob: Blob) -> Result<Id>;
    fn get(&self, id: Id) -> Option<&PlacedPoint>;
    fn remove(&mut self, id: Id) -> Option<PlacedPoint>;
}
```

```
6
7  pub trait Near {
8      fn near(&self, query: &Point, k: usize) -> Result<Vec<SearchResult
           >>;
9      fn add(&mut self, id: Id, point: &Point) -> Result<()>;
10 }
```

3.3 Adapters

Adapters implement ports for specific technologies:

Table 3: Available adapters.

Port	Adapter	Description
Place	MemoryStorage	In-memory hash map
Place	NVMeStorage	Memory-mapped files (planned)
Near	FlatIndex	Brute-force exact search
Near	HatIndex	Hierarchical Attention Tree

The HAT index adapter (published separately as `arms-hat`) demonstrates how domain-specific knowledge can be exploited for superior performance on hierarchical data.

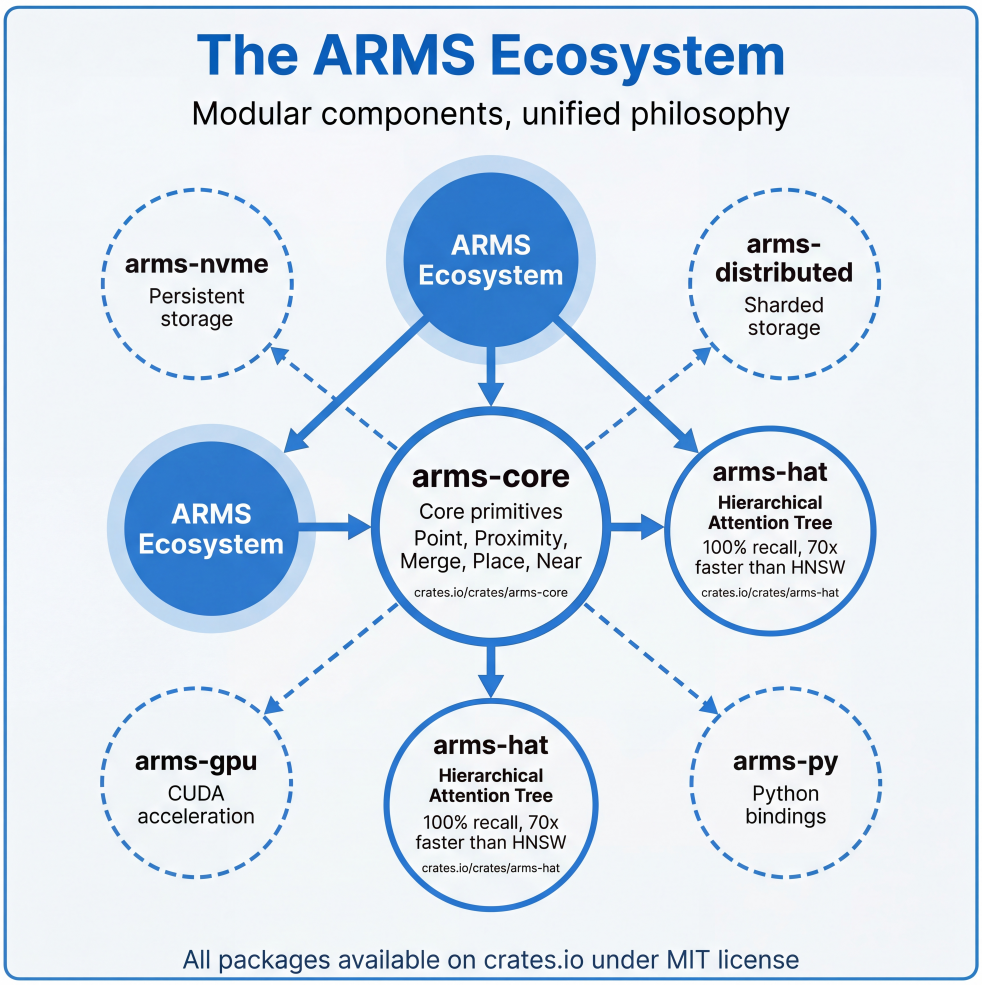

Figure 5: The ARMS ecosystem: `arms-core` provides the foundational primitives, while specialized adapters like `arms-hat` exploit domain-specific structure. Future adapters will add persistence, distribution, and GPU acceleration.

4 Position IS Relationship

The core philosophical innovation of ARMS is treating position as the fundamental relationship primitive.

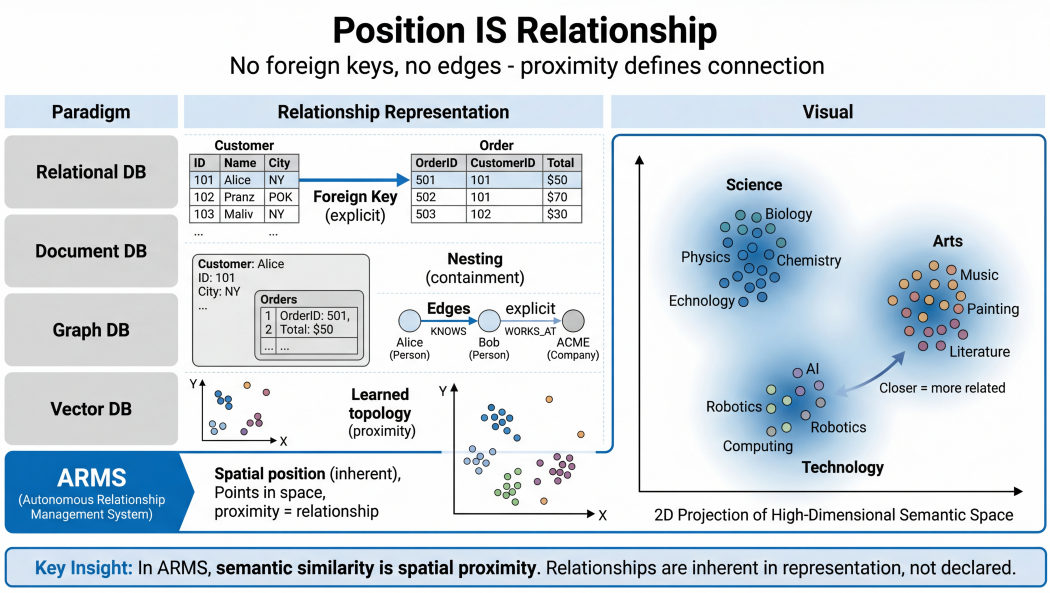

Figure 6: Position IS relationship: Comparison of relationship representation across database paradigms. ARMS uses spatial position as the fundamental relationship primitive, eliminating the need for explicit declarations.

4.1 Traditional Approaches

Table 4: Relationship representation in different paradigms.

Paradigm	Relationship	Limitation
Relational DB	Foreign keys	Must be declared explicitly
Document DB	Nesting	Limited to containment
Graph DB	Edges	Must be declared explicitly
Vector DB	Learned topology	Requires training/building
ARMS	**Spatial position**	**Inherent in representation**

4.2 Implications

When position is relationship:

1. **Schema-free**: No need to declare relationship types
2. **Continuous**: Relationships have degrees, not just existence
3. **Emergent**: New relationships discovered through proximity
4. **Composable**: Merged points represent group relationships

4.3 The Hippocampus Analogy

ARMS mirrors the function of the biological hippocampus:

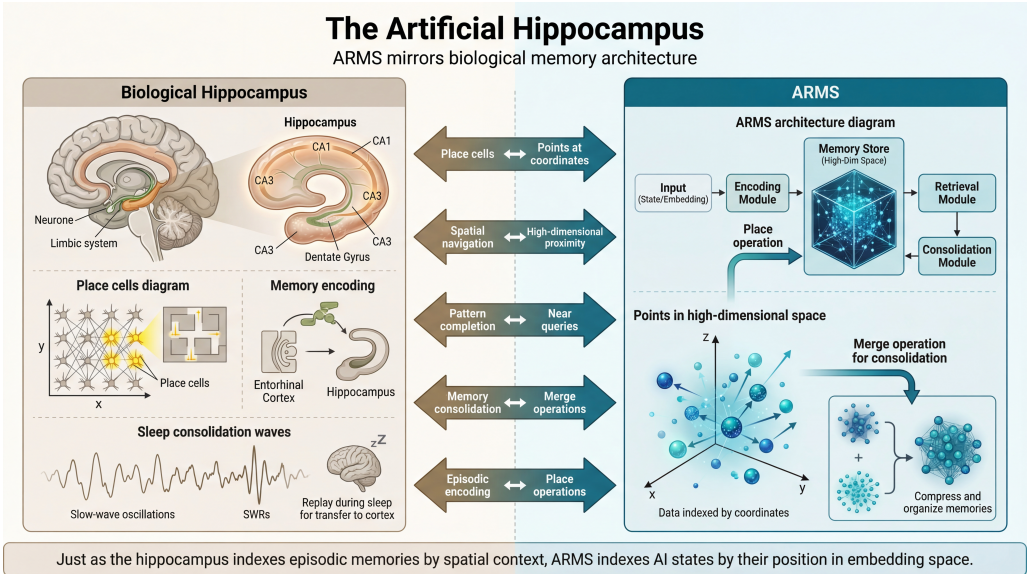

Figure 7: The hippocampus analogy: ARMS functions as an artificial hippocampus, enabling AI systems to form, consolidate, and retrieve episodic memories through spatial organization.

Table 5: Hippocampus vs ARMS.

Hippocampus	**ARMS**
Encodes episodic memories	Stores attention states
Spatial navigation	High-dimensional proximity
Pattern completion	Near queries
Memory consolidation	Merge operations
Place cells	Points at coordinates

5 Implementation

ARMS is implemented in Rust for performance and safety, with Python bindings planned.

5.1 Usage Example

Listing 4: Complete ARMS usage example.

```rust
use arms_core::{Arms, ArmsConfig, Point, Blob};

// Create ARMS with 768 dimensions
let mut arms = Arms::new(ArmsConfig::new(768));

// Store embeddings
let embedding = Point::new(vec![0.1; 768]);
let id = arms.place(embedding, Blob::from_str("hello")).unwrap();

// Query by proximity
let query = Point::new(vec![0.1; 768]);
let neighbors = arms.near(&query, 10).unwrap();

// Get with data
let results = arms.near_with_data(&query, 5).unwrap();
```

```
for (point, score) in results {
    println!("{}: {}", point.blob.as_str().unwrap(), score);
}
```

5.2 Performance

With the flat index (exact search):

Table 6: Flat index performance.

Points	Dimensions	Query Time
1,000	768	0.3ms
10,000	768	3ms
100,000	768	30ms

For large-scale deployments, the HAT index adapter provides $O(\log n)$ queries with 100% recall on hierarchical data.

6 Related Work

Vector Databases: Pinecone, Weaviate, Milvus, and Qdrant provide vector storage and retrieval. ARMS differs by providing a minimal primitive set and hexagonal architecture rather than a monolithic solution.

Memory-Augmented Networks: Neural Turing Machines and Differentiable Neural Computers use learned memory access. ARMS provides explicit, interpretable memory operations.

RAG Systems: Retrieval-Augmented Generation retrieves text for reprocessing. ARMS can store pre-computed attention states, avoiding recomputation.

Embedding Stores: LangChain, LlamaIndex provide embedding storage. ARMS provides lower-level primitives for building such systems.

7 Future Work

7.1 Planned Adapters

- **NVMe Storage**: Memory-mapped files for persistence
- **Distributed Storage**: Sharded across machines
- **GPU Index**: CUDA-accelerated similarity search

7.2 Applications

- **LLM Memory**: Long-term episodic memory for chatbots
- **Agent State**: Persistent state for AI agents
- **Attention Caching**: Store and retrieve KV cache states
- **Multimodal Memory**: Unified space for text, image, audio embeddings

8 Conclusion

ARMS provides a minimal, principled foundation for AI memory systems. By reducing memory operations to five primitives and adopting a hexagonal architecture, ARMS enables:

1. **Simplicity**: Five operations cover all memory needs
2. **Flexibility**: Swap storage, index, and API independently
3. **Performance**: Domain-specific adapters like HAT
4. **Philosophy**: Position IS relationship

ARMS functions as an artificial hippocampus for AI systems, enabling them to form, consolidate, and retrieve memories through spatial organization rather than explicit indexing.

Acknowledgments

I thank the open-source Rust community for excellent tooling and the researchers whose work on memory-augmented networks inspired this architecture.

References

A Code Availability

ARMS is available as open-source software:

- **Rust crate**: `arms-core` on crates.io
- **HAT adapter**: `arms-hat` on crates.io
- **Repository**: https://github.com/automate-capture/arms