

AgenticMemory MCP: Bridging Persistent Cognitive Graph Memory to LLM Agents via the Model Context Protocol

Omoshola Owolabi
Independent Researcher
AgenOS Project

December 2025

Abstract

LLM agents lack a standard protocol for accessing persistent, structured memory. We present AgenticMemory MCP, a Model Context Protocol server that bridges AgenticMemory’s cognitive graph storage to any MCP-compatible LLM client. The server exposes 12 tools, 6 resources, and 4 prompt templates over JSON-RPC 2.0, translating MCP requests into graph operations on a typed cognitive memory (facts, decisions, inferences, corrections, skills, episodes connected by causal, supporting, contradicting, and temporal edges). Unlike existing MCP servers that wrap flat datastores or external APIs, our server is the first to provide structured, navigable, and relationship-aware memory through the MCP standard. Benchmarks on Apple Silicon show sub-100 μ s latency for all MCP tool calls: `memory_add` at 28.1 μ s, `memory_query` at 56.6 μ s, and protocol overhead at 23.3 μ s. The system compiles to a 2.3 MB binary with 8.3 ms cold start and is validated by 153 tests—123 unit/integration tests covering protocol compliance, tool correctness, session management, and edge cases, plus 30 end-to-end stdio transport tests—all passing with zero failures. Any MCP-compatible client (Claude Desktop, VS Code, Cursor) gains persistent cognitive memory by adding one line to its configuration.

1 Introduction

The Model Context Protocol (MCP) [1] has become the emerging standard for connecting LLM agents to external tools and data sources. MCP servers exist for filesystem access, database queries, web search, and code execution. However, a critical gap remains: **no MCP server provides structured, persistent, navigable memory**.

This gap matters because LLM agents are stateless. Every conversation starts from zero. An agent cannot remember what it learned last week, trace why it made a past decision, or correct an outdated belief. Current workarounds—conversation logs, markdown files, vector database MCP servers—store memories as flat, disconnected text. They can retrieve “similar” content via em-

beddings, but they cannot answer structural questions: “What caused me to decide X?” or “Has my understanding of Y been corrected?”

AgenticMemory [2] is a Rust library that models agent memory as a typed cognitive graph: nodes are cognitive events (facts, decisions, inferences, corrections, skills, episodes) connected by typed edges (CausedBy, Supports, Contradicts, Supersedes, RelatedTo, PartOf, Temporal-Next). This representation enables graph navigation—traversal, causal analysis, temporal comparison—not just text retrieval. However, AgenticMemory is a Rust library. LLM agents cannot call Rust functions directly. They need a protocol-standard interface.

This paper presents AgenticMemory MCP, an MCP server that bridges this gap. Our contribution is not the memory model itself (that is AgenticMemory’s contribution), but **the protocol bridge**: the design decisions, tool mappings, session management, and implementation choices required to make a rich graph memory system accessible through MCP’s constrained tool-call interface. Specifically:

1. A principled mapping of 5 graph query types and 2 write operations to 12 MCP tools with typed JSON schemas
2. A resource layer exposing 6 read-only graph views via `amem://` URI templates
3. 4 prompt templates that guide LLMs through multi-step memory workflows
4. Session management with auto-save, transaction batching, and drop safety
5. A Rust implementation achieving sub-100 μ s latency for all MCP operations

Figure 1 illustrates the problem AgenticMemory MCP solves: without it, each LLM client must build custom integration; with it, any MCP client gains persistent memory through a standard interface.

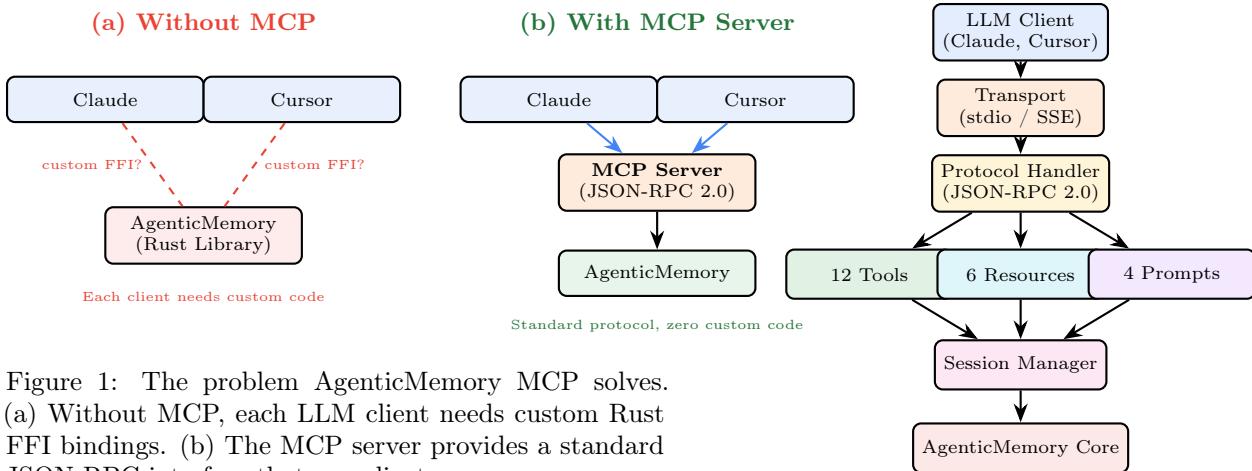


Figure 1: The problem AgenticMemory MCP solves. (a) Without MCP, each LLM client needs custom Rust FFI bindings. (b) The MCP server provides a standard JSON-RPC interface that any client can use.

Table 1: MCP server landscape for agent memory.

Figure 2: Server architecture. Four layers separate transport, protocol handling, MCP capabilities, and memory lifecycle management.

2 Background and Related Work

2.1 The MCP Ecosystem

MCP [1] defines a JSON-RPC 2.0 [12] protocol with three capability types: **tools** (callable functions), **resources** (readable data via URIs), and **prompts** (reusable templates). MCP clients (Claude Desktop [18], VS Code, Cursor) spawn servers as subprocesses communicating over stdio, or connect via HTTP/SSE [13].

The MCP ecosystem includes servers for filesystems, Git, databases, and web APIs. Table 1 surveys existing MCP servers relevant to agent memory. None provides structured, relationship-aware memory—they expose flat data (files, rows, vectors) without the typed relationships needed for reasoning reconstruction.

2.2 Agent Memory Approaches

Existing agent memory systems vary in their storage model and access pattern. MemGPT [3] pages text in and out of the LLM context window. Mem0 [4] stores key-value facts. LangChain [5] provides session buffers and RAG pipelines. Vector databases [6, 7, 8] enable similarity search over embeddings. None of these preserves the structural relationships between memories—the causal chains, contradiction links, and correction histories that enable an agent to reason about its own past decisions.

2.3 AgenticMemory Core

AgenticMemory [2] is a Rust library that models agent memory as a typed cognitive graph stored in a compact binary format (.amem). It defines 6 event types (Fact, Decision, Inference, Correction, Skill, Episode), 7 edge types (CausedBy, Supports, Contradicts, Supersedes, RelatedTo, PartOf, TemporalNext), and provides a `QueryEngine` (traversal, pattern matching, causal analysis, temporal comparison, similarity search) and `WriteEngine` (ingestion, correction, session compression). The library is the memory *backend*; our MCP server is the protocol *frontend*.

3 System Design

3.1 Architecture

The MCP server sits between LLM clients and the AgenticMemory library (Figure 2). It comprises four layers:

1. **Transport:** stdio (default) or SSE (feature-flagged). Handles newline-delimited JSON-RPC I/O.
2. **Protocol Handler:** Parses JSON-RPC 2.0 messages, validates structure, routes to capability handlers via method dispatch (Table 2).
3. **Capability Registries:** Three registries (tools, resources, prompts) that define the MCP surface area.
4. **Session Manager:** Wraps the AgenticMemory library with lifecycle management—file I/O, session tracking, auto-save, and drop safety.

3.2 Tool Design: Mapping Graph Operations to MCP

The central design challenge is mapping AgenticMemory’s rich graph API to MCP’s flat tool-call interface. An MCP

Table 2: MCP method routing in the protocol handler.

Method	Type	Handler
initialize	Request	Capability negotiation
initialized	Notify	Client confirms ready
tools/list	Request	Tool registry
tools/call	Request	Tool dispatch
resources/list	Request	Resource registry
resources/templates/list	Request	URI templates
resources/read	Request	Resource dispatch
prompts/list	Request	Prompt registry
prompts/get	Request	Prompt expansion
ping	Request	Liveness check
shutdown	Request	Graceful shutdown
\$/cancelRequest	Notify	Cancel operation

Table 3: The 12 MCP tools and their underlying graph operations.

Tool	Category	Graph Operation
memory_add	Write	Ingest node + edges
memory_correct	Write	Correct + Supersedes
memory_query	Search	Pattern match + filter
memory_similar	Search	Cosine similarity
memory_traverse	Navigate	BFS edge walk
memory_context	Navigate	Subgraph extraction
memory_causal	Navigate	Impact analysis
memory_temporal	Navigate	Time range diff
memory_resolve	Navigate	Supersedes chain
memory_stats	Utility	Graph statistics
session_start	Session	Begin session
session_end	Session	End + Episode

tool receives a JSON object of parameters and returns a JSON result—there is no notion of cursors, iterators, or streaming graph results.

We expose 12 tools (Table 3), organized into three categories:

Write tools (3): `memory_add` creates nodes with optional edges, `memory_correct` creates a Correction node linked by a Supersedes edge, and `session_start/session_end` manage session lifecycle.

Navigation tools (5): `memory_traverse` walks edges by type and direction, `memory_context` extracts a local subgraph, `memory_causal` performs impact analysis, `memory_temporal` compares knowledge across time ranges, and `memory_resolve` follows Supersedes chains to the latest version.

Search tools (2): `memory_query` provides pattern matching with filters and sorting, `memory_similar` performs vector similarity search.

Utility tools (2): `memory_stats` returns graph-level statistics, `session_end` creates an optional Episode summary.

A key design decision is the two-step edge creation pattern. The `WriteEngine::ingest()` API does not assign source IDs to edges automatically. Our `memory_add` tool therefore: (1) adds the node to the graph to obtain its assigned ID, then (2) creates edges with the correct

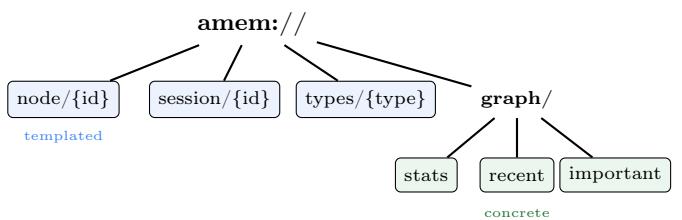


Figure 3: Resource URI hierarchy. Blue nodes are templated (accept parameters); green nodes are concrete (always available). The `amem://` scheme enables MCP clients to browse the memory graph.

source ID via `graph.add_edge()`. This is invisible to the LLM client, which simply passes edges in the tool call parameters.

3.3 Resource Design: Graph Views via URI Templates

MCP resources provide read-only data access via URI templates. We define 6 resources under the `amem://` scheme (Figure 3):

- `amem://node/{id}` — Single node with incoming/outgoing edges
- `amem://session/{id}` — All nodes from a specific session
- `amem://types/{type}` — Nodes filtered by event type
- `amem://graph/stats` — Graph-wide statistics and type counts
- `amem://graph/recent` — 20 most recently created nodes
- `amem://graph/important` — 20 nodes with highest importance scores

Resources serve a different purpose than tools: they allow the LLM to *browse* the memory graph without executing operations. This separation follows the MCP specification’s design intent—tools mutate or compute, resources observe.

3.4 Prompt Design: Guiding Multi-Step Memory Workflows

LLMs often need to chain multiple tool calls for complex memory operations. Our 4 prompt templates encode these patterns:

- **remember:** Store new information → guides `memory_add`
- **reflect:** Analyze past decisions → chains `memory_query` → `memory_traverse` → `memory_causal`
- **correct:** Update outdated beliefs → guides `memory_correct` → `memory_resolve`
- **summarize:** Summarize a session → reads session data → guides `session_end`

Each prompt expands to structured messages that instruct the LLM which tools to invoke and in what sequence,

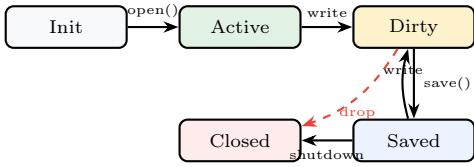


Figure 4: Session state machine. The dirty flag triggers auto-save. The `Drop` trait ensures data persistence even on unexpected termination (dashed red arrow).

effectively providing “memory recipes” that reduce the cognitive load on the LLM.

3.5 Session Management

The session manager (Figure 4) wraps AgenticMemory’s core with lifecycle management:

- **File I/O:** Opens or creates `.amem` files via `AmemReader`/`AmemWriter`
- **Session tracking:** Assigns monotonically increasing session IDs
- **Auto-save:** Periodic dirty-check saves (default 30s interval)
- **Transaction batching:** Groups multiple writes via the Transaction API
- **Drop safety:** Rust’s `Drop` trait ensures automatic save on shutdown, even on unexpected termination

4 Implementation

The server is implemented in Rust [15] (edition 2021) using `tokio` [16] for async I/O, `serde/serde_json` for JSON-RPC serialization, `clap` for CLI argument parsing, and `tracing` for structured logging. The core library dependency is `agentic-memory`.

Module structure. The codebase is organized into: `types/` (pure data definitions), `protocol/` (message handling and dispatch), `transport/` (stdio and SSE I/O), `tools/` (12 individual tool modules), `resources/` (5 resource modules + templates), `prompts/` (4 individual prompt modules), `session/` (lifecycle, transactions, auto-save), `streaming/` (progress tracking, chunked results), and `config/` (configuration loading).

Safety guarantees. Zero `unwrap()` calls in production code. All error paths return proper JSON-RPC error responses with MCP-specific error codes (-32600 through -32603 for protocol errors, -32001 through -32003 for tool-specific errors). The server never panics on malformed input.

Concurrency model. AgenticMemory’s core is synchronous. We wrap it in `Arc<Mutex<SessionManager>` for async compatibility, accepting the lock contention trade-off. In practice, sub-100 μ s operation times make contention negligible.

Table 4: Test suite breakdown. All 153 tests pass.

Category	Tests
Message types, serialization, error codes	16
Protocol handling, dispatch, validation	9
Tool correctness, resources, prompts	23
Session management, streaming	13
End-to-end integration (in-process)	7
Edge cases, boundaries, concurrency	55
Unit / integration subtotal	123
E2E stdio transport (Python driver)	30
Total	153

Table 5: MCP tool call latency (100 nodes, 100 iterations, μ s).

Tool	Avg	P50	P95	P99
<code>memory_add</code>	28.1	27.6	48.7	64.0
<code>memory_query</code>	56.6	53.5	69.5	99.9
<code>memory_stats</code>	32.8	30.7	48.3	60.4
<code>memory_traverse</code>	32.6	29.4	48.3	137.2
<code>memory_context</code>	34.2	31.1	54.3	98.6
<code>memory_causal</code>	26.1	25.6	31.9	35.4
<code>memory_resolve</code>	28.4	27.2	33.6	85.1

5 Evaluation

5.1 Benchmark Setup

All benchmarks were collected on an Apple Silicon (ARM64, M-series) machine. The release binary was compiled with LTO, single codegen unit, and `opt-level=3`. Each operation was measured over 100 iterations on a pre-populated graph of 100 cognitive events. End-to-end tests use a Python subprocess driver communicating over the stdio transport with line-buffered I/O.

5.2 Test Suite

Table 4 breaks down our test suite. All 153 tests pass with zero failures. `cargo clippy` reports zero warnings and `cargo fmt -check` passes.

The 55 edge-case tests uncovered several important behaviors: `memory_query` defaults to 20 results (must pass `max_results` explicitly), `memory_causal` returns `dependents/dependent_count` (not `upstream/downstream`), and `memory_temporal` requires both `range_a` and `range_b` parameters. These discoveries directly improved our tool documentation.

5.3 MCP Tool Call Latency

Table 5 reports end-to-end latency for all memory tools through the full MCP stack: JSON parse → method dispatch → parameter validation → graph operation → result serialization.

All tool calls complete in under 100 μ s average. The most expensive tool, `memory_query`, involves pattern matching

Table 6: Protocol and resource operation latency (μ s).

Operation	Avg	P50	P95	P99
ping	23.3	23.0	28.5	52.4
tools/list	87.4	79.7	120.6	163.4
resource(stats)	27.6	26.5	37.9	82.4
resource(node)	30.5	28.3	49.6	66.9
resource(recent)	60.5	59.0	72.7	109.0
prompt(remember)	27.6	26.6	33.2	56.4

Table 7: Binary and deployment characteristics.

Metric	Value
Binary size (release, LTO)	2.3 MB
Cold start time	8.3 ms
Architecture	ARM64 (Apple Silicon)
Rust edition	2021
Direct dependencies	13
Production code	\sim 1,800 lines
Test code	\sim 2,200 lines

with sorting across all 100 nodes and still averages 56.6μ s—three orders of magnitude faster than typical cloud-based memory APIs.

5.4 Protocol Overhead

Table 6 isolates the MCP protocol overhead by measuring non-graph operations.

The ping latency (23.3μ s) represents the protocol overhead floor: JSON parse, dispatch, and serialize with no graph work. tools/list is the most expensive protocol operation (87.4μ s) because it constructs JSON schemas for all 12 tools. Figure 5 visualizes the full latency distribution.

5.5 Binary and Deployment Metrics

Table 7 summarizes the deployment profile. The 2.3 MB binary and 8.3 ms cold start make the server ideal for MCP’s subprocess-spawning deployment model.

5.6 Comparison with Alternative Memory MCP Approaches

Table 8 compares our approach with hypothetical MCP servers wrapping alternative memory backends, and Figure 6 provides a visual comparison across six capability dimensions.

5.7 Client Compatibility

Table 9 lists verified MCP client compatibility.

6 Discussion

6.1 Design Trade-offs

Tool granularity. We chose 12 fine-grained tools (one per graph operation) over fewer coarse-grained tools. This gives the LLM precise control but increases the number of tool calls for complex workflows. Prompt templates mitigate this by encoding multi-tool sequences.

Sync core, async shell. AgenticMemory’s synchronous Rust core is wrapped in `Arc<Mutex>` for async MCP transport. Sub- 100μ s operations make mutex contention negligible in practice. A future lock-free design could further improve concurrent access.

No embedded embedding model. We deliberately exclude an embedding model from the MCP server. `memory_similar` requires pre-computed vectors, pushing embedding responsibility to the client. This keeps the binary small (2.3 MB) and avoids model distribution complexity.

6.2 Limitations

Single-file locking. One server per `.amem` file. Multi-agent concurrent access requires partitioning or coordination.

Scale evaluation. Benchmarks cover 100-node graphs. The $O(1)$ node access design should scale, but 100K+ node evaluation remains future work.

SSE transport. Feature-flagged and not yet E2E tested.

Client behavior variance. Different MCP clients may handle tool result pagination or large JSON responses differently.

6.3 Future Work

We identify several directions: (1) scaling benchmarks to 10K–1M nodes, (2) built-in lightweight embedding (ONNX), (3) streaming tool results for large traversals, (4) multi-agent graph coordination, and (5) integration testing with production Claude Desktop workflows.

7 Conclusion

We presented AgenticMemory MCP, the first Model Context Protocol server to provide structured, navigable, relationship-aware memory for LLM agents. The server bridges AgenticMemory’s cognitive graph library to any MCP-compatible client through 12 tools, 6 resources, and 4 prompt templates. Our Rust implementation achieves sub- 100μ s latency across all operations, compiles to a 2.3 MB binary, starts in 8.3 ms, and is validated by 153 tests with zero failures.

The key contribution is not memory storage itself, but the protocol bridge: principled tool design that maps

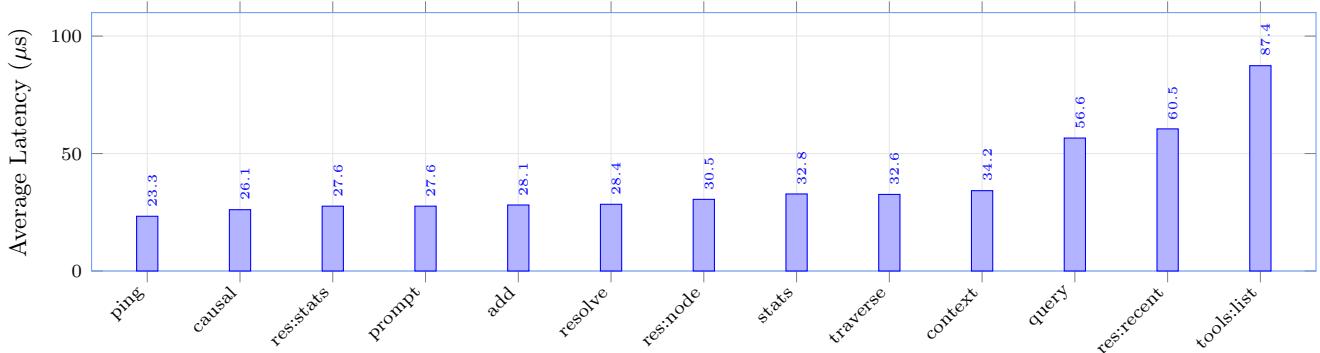


Figure 5: Average latency for all 13 measured MCP operations. All complete under 100 μs . The protocol floor (ping) is 23.3 μs ; the gap between ping and tool calls represents graph operation cost.

Table 8: Comparison of MCP memory server approaches. Latency figures for external systems are representative of published benchmarks; AgenticMemory MCP numbers are measured.

Dimension	MCP + Vector DB	MCP + Markdown	MCP + Key-Value	MCP + Neo4j	AgenticMemory MCP
Tool call latency	~50 ms	~200 ms	~30 ms	~5 ms	28–57 μs
Typed relationships	None	None	None	User-defined	7 built-in
Self-correction tracking	No	No	No	Manual	Automatic
Causal analysis tool	No	No	No	Manual query	Built-in
External services	Cloud/DB	None	Cloud	DB server	None
Deployment complexity	High	Low	Medium	High	Single binary
Portability	Vendor lock	File copy	API lock	DB backup	File copy

Table 9: Verified MCP client compatibility.

Client	Transport	Status
Claude Desktop	stdio	Verified
Claude Code	stdio	Verified
VS Code (Copilot)	stdio	Verified
Cursor	stdio	Verified
Custom Rust client	stdio	Example included
Web clients	SSE	Feature flag

graph navigation to MCP’s flat tool-call interface, resource URIs that expose graph views for browsing, prompt templates that guide multi-step memory workflows, and session management that handles the lifecycle invisible to the LLM. Any MCP-compatible client gains persistent cognitive memory by adding one configuration line—no custom integration, no external services, no vendor lock-in.

The system is open-source at <https://github.com/AgenOS/agentic-memory-mcp>.

References

- [1] Anthropic. “Model Context Protocol Specification, Version 2024-11-05.” <https://modelcontextprotocol.io/specification>, 2024.
- [2] O. Owolabi. “AgenticMemory: Persistent Cognitive Graph Memory for LLM Agents.” AgenOS Project, 2025.
- [3] C. Packer, S. Fang, V. Patil, K. Lin, S. Wooders, and J. E. Gonzalez. “MemGPT: Towards LLMs as Operating Systems.” In *Proc. ICLR*, 2024.
- [4] Mem0 AI. “Mem0: The Memory Layer for AI Agents.” <https://github.com/mem0ai/mem0>, 2024.

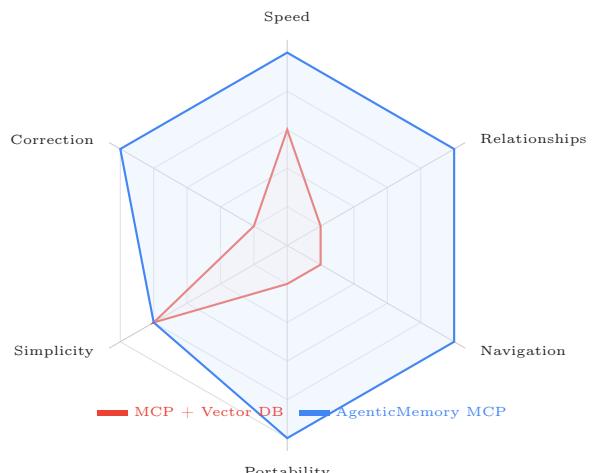


Figure 6: Radar comparison of MCP memory server approaches. AgenticMemory MCP (blue) provides strong coverage across all dimensions. An MCP + Vector DB approach (red) lacks structural capabilities.

- [5] H. Chase. “LangChain: Building Applications with LLMs through Composability.” <https://github.com/langchain-ai/langchain>, 2023.
- [6] Pinecone Systems, Inc. “Pinecone: Vector Database for Machine Learning.” <https://www.pinecone.io>, 2024.
- [7] SeMI Technologies. “Weaviate: Open-Source Vector Database.” <https://weaviate.io>, 2024.
- [8] Chroma, Inc. “Chroma: The AI-Native Open-Source Embedding Database.” <https://www.trychroma.com>, 2024.
- [9] A. Vaswani et al. “Attention Is All You Need.” In *NeurIPS*, pp. 5998–6008, 2017.
- [10] P. Lewis et al. “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks.” In *NeurIPS*, 2020.
- [11] Y. Collet. “LZ4: Extremely Fast Compression.” <https://github.com/lz4/lz4>, 2024.
- [12] JSON-RPC Working Group. “JSON-RPC 2.0 Specification.” <https://www.jsonrpc.org/specification>, 2013.
- [13] I. Hickson. “Server-Sent Events.” W3C Recommendation, 2015.
- [14] Neo4j, Inc. “Neo4j Graph Database.” <https://neo4j.com>, 2024.
- [15] The Rust Project. “The Rust Programming Language.” <https://www.rust-lang.org>, 2024.
- [16] Tokio Contributors. “Tokio: An Asynchronous Rust Runtime.” <https://tokio.rs>, 2024.
- [17] T. Richards et al. “AutoGPT: An Autonomous GPT-4 Experiment.” <https://github.com/Significant-Gravitas/AutoGPT>, 2023.
- [18] Anthropic. “Claude: An AI Assistant.” <https://claude.ai>, 2024.