WHITEPAPER

The Structured Context Layer

A Multi-Model Database Architecture
for Production AI Agent Systems

SurrealDB, Inc.

March 2026

Executive summary

AI agents fail in production because of context, not models. The infrastructure between enterprise data and the AI model - the context layer - must live in the database, not above it as middleware and not inside an analytical platform as a bolt-on feature. This paper presents the SurrealDB agent infrastructure stack: a vertically integrated system spanning SurrealDS (distributed ACID storage on cloud object storage), SurrealDB (a multi-model database unifying documents, graphs, vectors, and time-series in a single engine), and Spectron (structured agentic memory with entity extraction, knowledge graphs, and temporal fact tracking). No other product in the market owns this complete vertical. From object storage to agent memory - one stack, one transaction, full context.

Abstract

The transition from stateless language model interactions to persistent, autonomous AI agent systems has exposed a fundamental infrastructure gap. Agents fail in production not because models lack reasoning capacity, but because they lack reliable, structured, real-time context. Retrieval-augmented generation (RAG) over flat vector stores - the dominant approach since 2023 - suffers from semantic fragmentation, temporal blindness, and relevance degradation at scale. Memory middleware that abstracts over multiple specialist databases introduces consistency gaps, latency overhead, and semantic drift between stores. We argue that the context layer for AI agents must live *in the database* - not above it as middleware, not inside an analytical platform as a bolt-on feature - and that the database must be architecturally multi-model: documents, graphs, vectors, and time-series as native primitives in a single ACID-compliant transactional engine.

We present the SurrealDB agent infrastructure stack: a vertically integrated system spanning distributed ACID storage on cloud object storage, a multi-model database engine, and a structured agentic memory layer. This architecture eliminates the fragmented memory tax imposed by specialist database stacks, provides transactional consistency across all data models in a single query, and scales from embedded single-node deployments to petabyte-scale distributed clusters with compute-storage separation. We examine why memory middleware approaches are architecturally insufficient for production agent systems, how multi-model unification enables composable retrieval that specialist systems cannot replicate, and why the infrastructure layer beneath the database determines whether the context layer can operate at enterprise scale.

1. Introduction

1.1 The context wall

The AI industry's trajectory from 2023 to 2026 has been defined by a growing recognition: the limiting factor for agent reliability is not model capability but context quality. Enterprises deployed increasingly capable language models and watched agents fail - not because the models couldn't reason, but because they had nothing reliable to reason *over* [1].

This problem has been termed the **context wall** [2]. Agents built on flat vector stores cannot disambiguate entities, cannot track relationships between concepts, and cannot reason over how knowledge evolves over time. Similarity search returns text fragments that are geometrically proximate in embedding space, but geometric proximity is not contextual relevance. The result is agents that hallucinate, contradict themselves across sessions, and fail to coordinate with each other in multi-agent deployments.

The market has converged on the term **context layer** to describe the missing infrastructure between raw enterprise data and the AI model [2]. Several architectural approaches are competing to fill this gap: memory middleware that abstracts over existing databases, specialist agent databases that combine graph and vector capabilities, and analytical platforms adding agent-oriented features. Each approach carries fundamental architectural limitations.

1.2 Why middleware is insufficient

The most widely deployed approach to agent memory treats it as a middleware abstraction. Products like Mem0 [3] provide a managed layer that extracts facts from conversations, stores them across multiple backend databases (typically a vector store, optionally a graph database, and a relational store), and retrieves relevant context on query.

This architecture hides the complexity of multi-system orchestration from the developer, but it does not eliminate it. When context changes in one underlying system, synchronization to others is not guaranteed. A document updates in the vector store, but its graph relationships remain stale. An entity is renamed in the graph, but its embeddings in the vector store still reference the old name. The agent reasons over contradictory information - not because the middleware failed, but because the consistency boundary it can enforce stops at the API layer, not the storage layer.

We call this the **fragmented memory tax**: the latency of multi-system reconstruction, the risk of semantic drift between stores, and the loss of structural dimensionality when relationships exist only as pointers between separate systems rather than as first-class queryable objects.

1.3 Why specialist agent databases are incomplete

A second class of solutions - specialist agent databases like HelixDB and HydraDB [4] - combines graph and vector capabilities in a single product. However, "single product" does not mean "single engine." HydraDB, for example, self-hosts a separate vector substrate (Chroma,

Qdrant, or similar) alongside its temporal knowledge graph, and fuses results from both systems at the application layer [4]. The graph and the vectors are distinct internal components with distinct consistency models. Self-hosting internalizes the frankenstack rather than eliminating it - the synchronization problem between graph and vector stores still exists, it is simply managed internally rather than across third-party services.

Beyond the internal fragmentation, these systems are architecturally scoped to a single use case: agent memory. They lack native document models with full transactional consistency across all data types. They lack enterprise governance primitives - row-level permissions, field-level security, schema-enforced access control. They lack built-in backend capabilities (authentication, API endpoints, real-time subscriptions) that production agent systems require. And critically, they lack a cloud-native storage layer - compute and storage are coupled, scaling requires provisioning more nodes, and disaster recovery is an operational procedure rather than an architectural property.

For a production agent system, memory is one requirement among many. The agent also needs access to application data, user profiles, product catalogs, and operational state - ideally in the same transactional system. A specialist memory database forces the organization to maintain yet another system alongside its operational database, reintroducing the fragmentation it was designed to eliminate.

1.4 Our thesis

We argue that the context layer for AI agents must satisfy four properties simultaneously:

1. **Multi-model unification** - Documents, graphs, vectors, time-series, and relational data must be native primitives in a single engine, queryable in a single statement, consistent in a single ACID transaction.
2. **Structured memory with temporal awareness** - Entity extraction, knowledge graph construction, and temporal fact tracking must operate over the unified data layer, not as a separate system.
3. **Cloud-native distributed storage** - The storage layer must separate compute from storage, back data with commodity object storage, and provide horizontal scaling, fault tolerance, and disaster recovery as architectural properties.
4. **Production infrastructure** - Authentication, access control, API endpoints, real-time subscriptions, and schema enforcement must be built into the database, not bolted on.

No single product in the current market addresses all four. This paper presents a system that does.

2. The architectural problem

2.1 Flat vector stores destroy context

The first generation of AI data infrastructure - vector databases like Pinecone, Chroma, and Weaviate - was built for a single operation: approximate nearest-neighbor search over embeddings. This works for simple retrieval-augmented generation, where the task is to find document chunks semantically similar to a query and inject them into a prompt.

It fails as the primary data layer for agents because of six structural properties:

Flat storage. Documents are chunked and embedded as isolated vectors. The original structure - hierarchy, sections, cross-references - is destroyed at ingestion. Retrieval returns text fragments with no structural context [5].

No relationships. There is no way to model how entities relate to each other. Two entities with the same name - "Mercury" the internal project, "Mercury" the chemical element, "Mercury" the planet - occupy the same region of vector space. Without entity types and relational context, disambiguation is impossible. Metadata filters cannot solve this because the ambiguity is semantic, not structural [6].

No temporal awareness. Vector stores have no concept of when a fact was observed or when it became invalid. They return what is *similar*, not what is *current*. An agent asking "Where does the user live?" may retrieve a five-year-old address alongside the current one, with no signal indicating which is valid.

Semantic fragmentation. Standard chunking destroys the connections between facts. Research suggests that roughly 40% of naively chunked content becomes "semantically invisible" - isolated from the entities and context it refers to [5]. If a user says "I hate that framework" and the framework ("React") was mentioned several chunks earlier, vector search can never map "that framework" to "React" because the isolated chunk carries no entity reference. This is the **orphaned reference problem**: pronouns, demonstratives ("that project," "the same client"), and implicit references lose their targets when text is split into independent vectors. The agent retrieves the sentiment but not the subject, or the subject but not the sentiment.

Vocabulary mismatch. A persistent failure mode in vector-only retrieval is the disconnect between user intent and stored content. A user asks "Why is the app behaving strangely?" The relevant memory contains "Error 503: Service Unavailable." Standard embedding models place these two strings far apart in vector space because they share no lexical or immediate semantic overlap. The user is expressing abstract intent; the memory records a technical fact. Vector retrieval alone cannot bridge this gap - it requires either the lexical precision of keyword search to catch "Error 503" or structural traversal to connect the user's application to the service that failed.

Relevance degradation at scale. Vector databases can maintain high recall with well-tuned indexes, but *relevance* deteriorates as the corpus grows [7]. More documents mean more noise in similarity results because geometric proximity is not the same as contextual relevance. Without structural signals - entity types, relationship constraints, temporal context - there is no

way to distinguish "similar text" from "actually relevant context."

2.2 The fragmented memory tax

The natural response to vector database limitations has been to add systems: a graph database for relationships, a document store for structured records, a relational database for application data, an authentication service, an API framework, a real-time messaging layer. This "frankenstack" approach solves individual capability gaps but introduces systemic problems:

Latency of reconstruction. Every time an agent needs to "remember" something, it makes multiple asynchronous calls to different APIs, waits for responses, and uses application logic to join the results. In an autonomous agent loop running at millisecond speed, these round-trips compound into seconds of latency.

Semantic drift. When a document updates in one store but its vector embedding in another store is not refreshed, or its graph relationship remains stale, the agent reasons over contradictory information. Different agents operating on different definitions of the same entity is the enterprise-scale version of this problem.

Loss of dimensionality. A standalone vector search can tell an agent that "Product A is similar to Product B," but it cannot explain *why* (the relationship type, the business constraint, the version history) without a complex multi-stage join across multiple systems.

Consistency boundaries. Each system has its own consistency model. The vector store may be eventually consistent while the graph database provides strong consistency. The agent's view of the world depends on which system it queries and when - a source of non-deterministic behavior that is invisible in development and catastrophic in production.

Memory middleware products abstract over this fragmentation, providing a cleaner developer experience. But the abstraction hides the complexity; it does not eliminate it. The consistency boundaries remain. The semantic drift persists. The latency overhead compounds. The middleware layer itself becomes another system to operate, monitor, and debug.

2.3 The read-think-write loop

An AI agent operates through a continuous cycle: perceive the environment, reason over memory, commit an action or observation back to storage. This loop runs in milliseconds, and it never stops [2].

Modern data platforms - Snowflake, Databricks, BigQuery - are optimized for analytical throughput: answering questions about the past at scale. Agents don't need to analyze the past; they need to act in the present. The distinction is design intent:

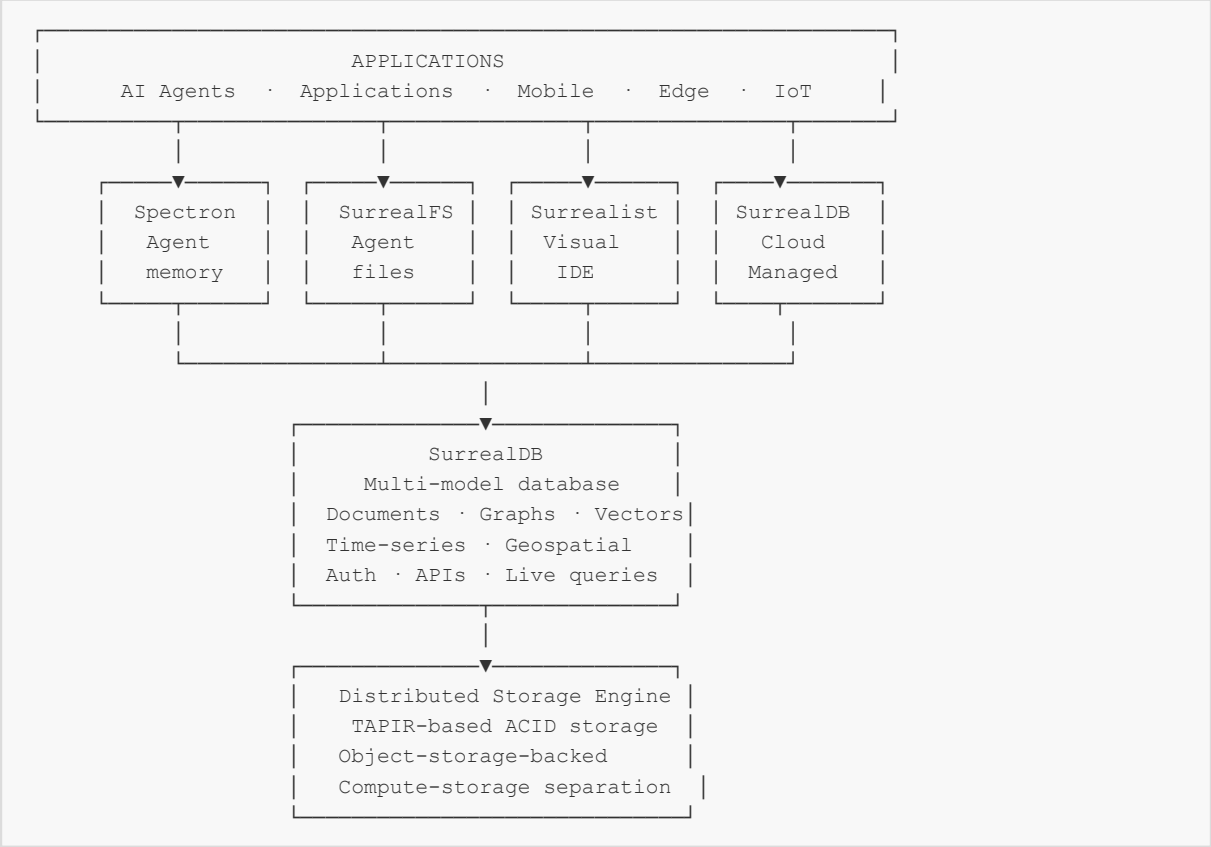
- **Data platforms** answer "what happened" - optimized for large analytical scans over historical data.

- **The context layer** answers "what should happen next" - optimized for low-latency, transactional read-think-write loops over current state and evolving knowledge.

This is why the context layer must live *in the database*, not above it as middleware. Only at the database layer can context be kept consistent, governed, secured, and transactionally unified with the canonical knowledge that grounds it. Middleware cannot enforce transactional consistency across the systems it orchestrates. Analytical platforms cannot provide the sub-millisecond transactional latency that the agent loop demands.

3. Architecture

The SurrealDB agent infrastructure stack is a vertically integrated system with four layers:



3.1 Layer 1: Distributed ACID storage on object storage

The foundation of the stack is a next-generation distributed storage engine that fundamentally separates compute from storage. Data lives in commodity cloud object storage (S3, GCS, Azure Blob Storage). The database engine runs as stateless, elastic compute on top.

This represents a **Generation 3** database architecture [8]:

Generation	Architecture	Limitation

Gen 1 (MySQL, PostgreSQL self-hosted)	Compute and storage on one machine	Scale = bigger box
Gen 2 (Aurora, AlloyDB, Exadata)	Storage separated into proprietary cloud tiers	Data locked in. Single vendor. Single engine.
Gen 3 (SurrealDB distributed, Databricks Lakebase)	Data in commodity object storage. Stateless elastic compute	Open architecture. Cross-cloud.

3.1.1 TAPIR transaction protocol

The storage engine uses TAPIR (Transactional Application Protocol for Inconsistent Replication) [9] for distributed consensus. Each availability zone contains a dedicated TAPIR write node. Unlike leader-based replication protocols, writes are not funneled through a single leader - they scale horizontally across all write nodes in the cluster. Each transaction is encapsulated within a single TAPIR consensus decision at the writing node, allowing write throughput to grow linearly as nodes are added.

Transactions commit once a quorum of nodes acknowledges the writes. Majority agreement is sufficient; individual node failures do not block commits. This delivers lower latency and higher throughput than leader-based replication while maintaining strict ACID guarantees.

3.1.2 Object-storage-backed persistence

All data lives in commodity cloud object storage. LSM tree layers are synced to object storage after range compaction. Transaction logs are written asynchronously for durable recovery. This architecture provides:

- **Infinite storage scale** at a fraction of provisioned disk cost.
- **99.999999999% durability** inherited from the underlying cloud platform.
- **Reduced cross-availability-zone costs.** Traditional distributed databases replicate data directly between nodes across AZs - expensive cross-AZ network traffic that compounds at scale. The SurrealDB storage engine routes data through object storage instead, significantly reducing cross-AZ traffic.

3.1.3 Compute-storage separation

The database engine runs as stateless compute. This enables:

- **Scale to zero.** When no queries are running, no compute is billed. Development, staging, and idle production environments cost nothing.
- **Elastic scaling.** Read proxy nodes can be added or removed independently of write nodes. Compute scales with demand, not ahead of it.
- **Instant branching.** Clone a petabyte-scale database in seconds - for development, testing, experimentation, or A/B evaluation. The underlying data in object storage is shared; only metadata diverges.
- **Instant recovery.** A crashed node restores from object storage and replays the recent transaction log. No streaming gigabytes from peer nodes. Recovery time is independent of

dataset size.

- **Native disaster recovery.** A zone or region failure means pointing new nodes at the same object-storage bucket and replaying the log. No separate backup infrastructure. No snapshot schedules. The object store *is* the durable tier.

3.1.4 Why the storage layer matters for agent memory

This infrastructure layer is invisible to the agent - but it determines whether the context layer can operate at enterprise scale. Every capability built on top of SurrealDB inherits the storage engine's properties:

- Agent memory (Spectron) scales to zero when agents are idle and scales elastically when they're active.
- Knowledge graphs accumulate petabytes of structured memory without hitting storage ceilings.
- Multi-region agent deployments replicate memory through object storage, avoiding cross-AZ network costs.
- An entire agent's knowledge graph can be branched in seconds for testing, experimentation, or evaluation - a workflow impossible in systems with coupled compute and storage.
- Agent memory survives infrastructure failures because the object store is the durable tier.

No specialist agent database or memory middleware product owns its storage layer. They all depend on whatever backend they happen to use. The SurrealDB stack owns the storage layer, and that storage layer is Gen 3.

3.2 Layer 2: Multi-model database engine

SurrealDB is a multi-model database built in Rust that unifies document, graph, relational, time-series, geospatial, key-value, and vector data in a single engine. All data models are native primitives - not additive features bolted onto a single-model core. This distinction is critical: in SurrealDB, a graph edge *is* a document. It can hold the weight of a relationship, the timestamp of an interaction, the vector embedding of the context, and arbitrary structured metadata - all retrievable in a single query.

3.2.1 SurrealQL: composable multi-model queries

The query language, SurrealQL, enables composable retrieval across all data models in a single statement:

- **HNSW vector similarity** for semantic ranking over embeddings.
- **BM25 full-text search** for keyword-based candidate selection.
- **Graph traversal** over typed edges for relational context and multi-hop reasoning.

- **Bi-temporal filtering** with native support for two independent time dimensions - transaction time (when data was recorded) and valid time (when data was true in the real world) - enabling queries across both the history of the database and the history of the domain.
- **Schema constraints** for type-safe entity resolution and ontological enforcement.

These capabilities compose in a single query, within a single ACID transaction. There is no multi-system orchestration. There is no consistency gap between the vector result and the graph traversal - they are resolved from the same transactional snapshot.

This composability is architecturally impossible in systems where graph and vector capabilities live in separate engines, even when orchestrated by middleware. The fusion of retrieval signals must happen at the storage layer to guarantee consistency; any fusion that happens above the database layer is subject to the consistency boundaries of the underlying systems.

3.2.2 Schema as ontology

SurrealDB's schema system acts as an ontology layer. Entity types are defined with typed fields. Relationships between entity types are constrained using typed graph edges. The *kinds* of entities and the *valid relationships* between them are part of the database schema - not conventions in application code.

This provides several properties that ad-hoc entity extraction cannot:

- **Structural constraints.** A relationship between a `person` and a `company` must be of type `works_at`, `advises`, or `founded` - not arbitrary free text. Invalid relationships are rejected at write time.
- **Schema evolution.** When the ontology changes (a new entity type is introduced, a relationship constraint is relaxed), the change is applied at the database level and is immediately visible to all consumers.
- **Deterministic disambiguation.** When an agent encounters an ambiguous entity name, schema constraints narrow the space of valid interpretations. "Mercury" the `project` and "Mercury" the `chemical_element` are structurally distinct entity types, resolvable without probabilistic inference.

3.2.3 Rich edges: relationships as documents

In a traditional graph database, an edge is a simple pointer between two nodes - a label and perhaps a few properties. In SurrealDB, an edge is a full document. It can carry:

- The semantic type of the relationship (e.g., `depends_on`, `prefers`, `caused_by`).
- The timestamp of the interaction or observation.
- The vector embedding of the context in which the relationship was established.
- Arbitrary structured metadata: confidence scores, source references, provenance.

This means an agent can traverse a relationship and find, embedded directly within that link,

everything it needs to reason about the connection. No separate lookup to a document store for metadata. No call to a vector store for embeddings. One operation, one complete thought.

Consider an agent monitoring a production system that receives an alert about a failing API endpoint. With a vector database, the agent searches for documentation semantically similar to the error message - useful, but shallow. With SurrealDB, the agent traverses from the failing endpoint to the upstream service, to the team that owns that service, to the on-call engineer, and simultaneously retrieves the vector-embedded runbook most relevant to this specific failure mode - all in a single query. The structural traversal and the semantic search are one atomic operation.

3.2.4 Graph-derived inference: reasoning from topology

A property of graph-structured memory that flat indexes cannot replicate is the ability to derive conclusions from the *topology* of relationships - conclusions that were never explicitly stated in any single document or conversation.

Consider a user who, across multiple unrelated sessions, rejects recommendations involving proprietary cloud vendors, accepts recommendations involving open-source tooling, and expresses cost sensitivity in budget discussions. In a vector store, these are three unrelated chunks in different regions of embedding space. No query will retrieve all three unless it happens to be semantically similar to all of them.

In SurrealDB's graph model, these interactions produce a subgraph of typed edges: `user --[rejected]--> vendor_a`, `user --[rejected]--> vendor_b`, `user --[prefers]--> open_source`, `user --[optimizes_for]--> cost_efficiency`. The topology of this subgraph encodes a vendor preference that was never explicitly stated - the user never said "I prefer open-source over proprietary," but the pattern of edges makes it structurally recoverable.

This capability extends beyond preferences. Outcome signals - whether a recommendation was acted upon, whether a plan succeeded, whether a decision was later reversed - can be encoded as edge-level annotations on the graph. A `recommended` edge can carry a `followed: true` annotation; a `decided` edge can carry a `reversed: 2024-06` annotation. Over time, the graph accumulates not just what the user said, but what they *did* - transforming the memory layer from a passive record of stated facts into an active model of demonstrated behavior.

Because SurrealDB's edges are full documents, these annotations are not key-value properties on a pointer - they are structured records with their own metadata, temporal context, and vector embeddings. An agent can traverse the preference subgraph, retrieve the annotations, and reason over the accumulated evidence in a single query.

3.2.5 Built-in production infrastructure

SurrealDB ships with infrastructure that production agent systems otherwise require as separate services:

- **Authentication and access control.** `DEFINE ACCESS` configures authentication methods (JWT, token, OAuth) at the database level. Row-level permissions are declared in the schema and enforced at the query layer.
- **API endpoints.** `DEFINE API` creates HTTP routes directly in SurrealQL. The database is the API server.
- **Real-time subscriptions.** Live queries push changes to agents the instant data mutates via WebSocket. No polling. No stale cache.
- **Event triggers.** `DEFINE EVENT` fires on data changes, enabling reactive workflows and audit trails without external message queues.
- **Protocol flexibility.** SurrealQL, ANSI-SQL via Postgres wire protocol, GraphQL, and WebSocket interfaces. Agents connect through whatever protocol their framework supports.
- **Native MCP connectivity.** The SurrealDB ecosystem provides Model Context Protocol (MCP) servers across the entire platform - not just for the database. SurrealDB's MCP server exposes the full operational surface of the multi-model database: execute SurrealQL, traverse graphs, perform vector searches, explore schemas, and manage definitions through structured MCP tool calls. Spectron ships its own MCP server for memory ingestion, knowledge graph queries, and entity/fact retrieval. SurrealDB Cloud ships a third MCP server for infrastructure management - provisioning, scaling, and monitoring deployments. Any MCP-compatible client - AI-powered IDEs, agent frameworks (Claude, OpenAI, LangChain, CrewAI), or custom implementations - connects to the entire platform through the standard protocol. No custom SDKs, no integration code. An agent discovers and invokes capabilities across data, memory, and infrastructure through the same protocol it uses for every other tool in its environment.

3.3 Layer 3: Structured agentic memory (Spectron)

Spectron is a purpose-built agentic memory layer that gives AI agents persistent, structured, queryable memory. It combines vector search with a knowledge graph to store not just text chunks but entities, relationships, facts, and their temporal evolution.

Spectron is built entirely on SurrealDB. Memories, chunks, entities, predicates, vector indexes (HNSW), and the asynchronous job queue - all stored in SurrealDB tables. Spectron inherits every multi-model capability, every ACID guarantee, and every infrastructure property of the layers beneath it.

3.3.1 Ingestion pipeline

When an agent sends text to Spectron (conversations, documents, observations), an asynchronous pipeline processes the input without blocking the query path. Memories are enqueued to a job queue - itself backed by SurrealDB - and processed by a pool of concurrent ingestion workers:

1. **Chunking and embedding.** Text is segmented and embedded using HNSW-indexed vector representations (1536-dimensional, cosine similarity). Chunks are stored as SurrealDB documents with full metadata.

2. **Entity and fact extraction.** An LLM extracts entities and facts from the text in a single structured pass. The extraction prompt enforces identity resolution at ingestion time: *"Do not generate multiple entities for the same real-world object. If an entity is mentioned multiple times but referred to by different names or pronouns, always use the most complete identifier."* The result is a set of typed entities (each with a `name` and `type`) and subject-predicate-object facts linking them.

3. **Entity deduplication.** Before storage, extracted entities are deduplicated by their (type, name) pair. If the same entity has already been extracted from a previous memory, the existing entity node is reused rather than creating a duplicate. This ensures that the knowledge graph converges toward a single canonical node per real-world entity, regardless of how many memories mention it.

4. **Fact validation and storage.** Facts are validated against the extracted entity set - a fact cannot reference an entity that was not identified in the same extraction pass. Valid facts are stored as SurrealDB graph edges (`predicate` relations of type `RELATION IN entity OUT entity`), with the verb as a labeled edge and temporal metadata (`created_at` as an immutable timestamp, `invalid_at` as an optional datetime for superseded facts).

5. **Provenance tracking.** A `derived_from` relation links each entity back to the memory it was extracted from, with an observation timestamp. This creates a traceable provenance chain: for any entity in the knowledge graph, the system can identify which conversation or document it originated from and when.

6. **Entity embedding.** Entity names are independently embedded and indexed alongside memory chunks, enabling vector search to match queries against entities directly - not just against the text chunks that mention them.

The entire pipeline writes to SurrealDB in a single transactional system. The job queue, the memory chunks, the entity nodes, the fact edges, and the vector indexes all live in SurrealDB tables. There is no synchronization step between separate stores - ingestion is atomic.

3.3.2 Hybrid retrieval

On query, Spectron performs a multi-signal retrieval that combines four complementary search paths:

Vector similarity over memory chunks and entity embeddings. This captures semantic meaning - retrieving passages that are conceptually related to the query even when they share no lexical overlap. HNSW indexes provide sub-millisecond approximate nearest-neighbor search over high-dimensional embeddings.

BM25 full-text search for keyword-based candidate selection. This captures lexical precision - ensuring that rare but critical tokens (project IDs, error codes, usernames, technical identifiers) strongly influence retrieval. A query about "Error 503" must retrieve the memory containing that exact string, even if the embedding model places it far from the user's natural-language description of the problem.

Graph traversal over the fact graph for matched entities. When entities are identified in the query, Spectron traverses typed relationships to recover relational context that no single text chunk contains. This operates at two levels: first, direct entity matching retrieves facts about entities mentioned in the query; second, neighborhood expansion follows relationships from entities found in high-confidence vector results to surface adjacent context - related tasks, blockers, decisions, and outcomes that are structurally connected but may not be semantically similar.

Bi-temporal filtering to surface facts valid at the relevant point in time. SurrealDB's two time dimensions - transaction time (when data was recorded) and valid time (when data was true in the real world) - allow queries to resolve not just "what is current" but "what was true at time X" and "what did we believe at time Y." Queries referencing "last month," "before the migration," or "the current status" are resolved against valid-time metadata, filtering out superseded facts and ordering results by temporal relevance.

These four signals compose in a single SurrealQL query within a single ACID transaction. The result is a structured response:

- **Memories**: Relevant text passages ranked by a composite of semantic similarity and lexical match.
- **Facts**: Subject-predicate-object triples grouped by entity, with observation timestamps and relationship context.

Why composable retrieval solves the vocabulary mismatch problem. The vocabulary mismatch described in Section 2.1 - where user intent and stored content occupy distant regions of embedding space - is addressed by the composition of multiple retrieval signals, not by any single signal alone. Vector similarity captures semantic intent ("app behaving strangely"). BM25 captures the technical identifier ("Error 503"). Graph traversal connects the user's application to the service that generated the error. Temporal filtering ensures the result reflects the current state, not a resolved incident from last quarter. No single retrieval primitive can bridge this gap; the composition of all four, within a single transactional query, produces context that is simultaneously semantically relevant, lexically precise, structurally grounded, and temporally current.

In systems where these retrieval signals live in separate engines, the fusion happens at the application layer - subject to consistency boundaries, latency overhead, and the combinatorial complexity of merging results from different systems with different ranking models. In SurrealDB, the fusion happens at the query layer, within a single ACID transaction, resolved from a single transactional snapshot.

3.3.3 Disambiguation through structure

A persistent failure mode in vector-only retrieval is entity ambiguity. "Mercury" the internal project, "Mercury" the chemical element, and "Mercury" the planet occupy the same region of embedding space. Metadata filters (e.g., `category: "project"`) provide a partial solution but require the application to know which category to filter by - which is precisely the disambiguation problem.

Spectron resolves ambiguity through two complementary mechanisms. At **ingestion time**, the entity extraction prompt enforces canonical identity: the LLM is instructed to produce a single entity node per real-world object, using the most complete identifier available, and entities are deduplicated by (type, name) before storage. This means the knowledge graph structurally distinguishes `entity:project:Mercury` from `entity:element:Mercury` from the moment facts are ingested - not at query time.

At **query time**, the graph provides structural context: which entities named "Mercury" exist, what types they are, and what relationships connect them to other entities the agent has recently interacted with. If the user has been discussing the internal project, the graph's recent interaction edges make the correct "Mercury" structurally recoverable. Disambiguation is a graph operation at both ingestion and retrieval - not a probabilistic guess at either stage.

3.3.4 Temporal reasoning and the destructive update problem

A common pattern in memory systems is the **iterative resolution loop**: for every incoming fact, search for semantically similar existing facts, then ask an LLM whether to update or delete the old record. This approach has two dangerous failure modes:

- **False positive deletes.** "I love Python" and "I used to love Python" are semantically close but chronologically distinct. An LLM-based resolution step can purge historical context based on a probabilistic judgment, destroying information that explains *why* something changed.
- **Loss of the decision tree.** When a user moves from New York to London, an overwrite-based system records the new location and discards the old one. The agent can no longer answer "Why did you move?" or "Where did you live in 2022?" because the reasoning context was never preserved.

SurrealDB eliminates this entirely through **native bi-temporal versioning** - two independent time dimensions tracked at the database level:

- **Transaction time** (tracked automatically at the storage level) - Every version of every record carries the timestamp of when it was committed. The database maintains a complete, immutable, append-only history. No destructive updates. No lost audit trails.
- **Valid time** (declared at the data and query level) - Records carry the real-world date range during which they are considered valid. "The user lived in New York from 2020 to 2024" and "The user lives in London from 2024 onwards" coexist as explicit temporal facts.

At the schema level, Spectron's fact storage makes this concrete. The `predicate` table is defined as a SurrealDB `RELATION` - a typed graph edge from one entity to another - with `created_at` as a `READONLY` timestamp (immutable once committed) and `invalid_at` as an optional datetime (null for current facts, set when a fact is superseded). The schema enforces append-only semantics: once a fact is committed, its observation timestamp cannot be altered.

When a user's location changes, the knowledge graph does not overwrite the previous fact. Both facts persist with distinct valid-time ranges:

- `user:alice --[located_in]--> city:new_york` (valid: 2020–2024)

- `user:alice --[located_in]--> city:london` (valid: 2024–present)

The combination of both time dimensions enables temporal queries that no single-time-axis system can answer:

- "Where did the user live in 2022?" - valid time query, returns New York.
- "When did we first learn they moved to London?" - transaction time query, returns the commit timestamp.
- "What did we believe on March 1 about where the user would be living in June?" - bi-temporal query, resolving the database state as of March 1 against the valid-time range that includes June.

The full decision tree - what changed, when it changed, when we learned about it, and why - is always preserved and queryable. This is essential for regulatory compliance, financial reporting, and any agent memory system where temporal reasoning must be precise.

This approach aligns with the Git-style versioning model advocated in recent memory system research [4], but goes further. Systems that implement append-only graphs as an application-level pattern still rely on the application to manage temporal metadata correctly. SurrealDB's bi-temporal versioning is a database-level primitive - transaction time is tracked automatically at the storage layer, and valid time is a first-class query dimension. And because the temporal graph is stored in the same ACID-compliant transactional engine as the vector indexes, document records, and application data, temporal queries compose with vector similarity and structural traversal in a single statement. There is no separate temporal graph system to synchronize.

3.3.5 Solving semantic fragmentation at ingestion

The orphaned reference problem described in Section 2.1 - where naive chunking severs the connection between a reference and its target - is a failure that occurs at ingestion time and cannot be recovered at query time. If a chunk contains "I hate that framework" but the entity "React" exists only in a distant chunk, no amount of retrieval sophistication will reconnect them.

Spectron's entity extraction pipeline addresses this by resolving references at ingestion time. When text is processed, the LLM identifies entities and relationships in context - before the text is chunked and embedded. Extracted entities become first-class nodes in the knowledge graph, linked to the chunks that reference them. At query time, the agent traverses typed graph edges to explicit entities, not statistical guesses about which chunk might be related.

This is architecturally distinct from approaches that attempt to enrich chunks with surrounding context (sliding window enrichment). While contextual enrichment improves individual chunk quality, it remains fundamentally bounded by the window size and cannot recover references that span entire conversations. Graph-based entity resolution, by contrast, creates a persistent structural link between the reference and its target that survives regardless of chunk boundaries.

3.3.6 Preference accumulation and sentiment

Vector databases treat each session as stateless with respect to preference learning. Any personalization depends on retrieving the specific chunk that mentioned a preference - which fails when preferences are implicit, distributed across sessions, or expressed indirectly.

Spectron's knowledge graph accumulates preferences and outcomes as structured, typed relationships that persist and compound across the full interaction history. When a user repeatedly accepts recommendations involving a particular approach, declines alternatives, or expresses satisfaction or frustration, these signals produce graph edges that encode behavioral patterns as explicit relationships: `user --[prefers]--> approach_a`, `user --[avoids]--> approach_b`, `user --[optimizes_for]--> simplicity`.

Critically, Spectron preserves not just the fact of a preference but its **intensity and reasoning context**. When a user says "I absolutely hate React; it's a nightmare to debug," a vector store may preserve the factual content but lose the emotional weight. Spectron's entity extraction captures the relationship (`user --[dislikes]--> react`) with metadata that encodes the intensity and the rationale - stored as structured fields on the graph edge, not buried in a text chunk that may or may not be retrieved.

Over time, this accumulation transforms the memory layer from a record of what was said into a model of what the user values. An agent assembling context for a recommendation can traverse the preference subgraph and weight its suggestions based on accumulated evidence - not just the last thing the user mentioned, but the full pattern of demonstrated behavior across all sessions.

3.4 Layer 4: Persistent file memory (SurrealFS)

SurrealFS complements Spectron's structured memory with a file-and-folder abstraction for agent working state. Agents interact with a virtual file system using Unix-like commands (`ls`, `cat`, `write`, `edit`, `grep`) - the simplest possible memory model. All paths and content are stored in SurrealDB tables.

SurrealFS provides:

- **Working memory** - Running scratchpad notes updated each session.
- **Preference memory** - Learned user preferences organized in directory hierarchies.
- **Procedural memory** - Reusable knowledge base articles and workflow documentation.
- **Multi-agent shared memory** - When backed by a remote SurrealDB instance, multiple agents share the same virtual file system. One agent's writes are immediately visible to all others.

4. Multi-agent coordination

4.1 The blackboard pattern

In many multi-agent frameworks, agents communicate via direct message passing. This creates a "telephone game" where context is lost, versions diverge, and the source of truth becomes a moving target.

The SurrealDB stack enables the **blackboard pattern**: a shared, consistent source of truth where agents commit findings rather than telling each other what they found.

- **SurrealFS** provides a shared filesystem. One agent writes a customer summary; another reads it immediately.
- **Spectron** provides a shared knowledge graph. Entity extraction from one agent's conversations enriches the knowledge available to all agents. Facts converge rather than diverge.
- **SurrealDB** provides transactional consistency with live queries that push changes to agents the moment they occur.

4.2 The agentic race condition

Without ACID guarantees at the database layer, multi-agent systems suffer from a synchronization failure we term the **agentic race condition**: Agent A acts on information that Agent B is currently modifying. In a fragmented architecture, this synchronization gap is where reasoning breaks down.

SurrealDB eliminates this with snapshot isolation. Each agent reasons over a consistent transactional view of the database. Committed writes from one agent are immediately visible to subsequent transactions from other agents. There is no synchronization gap across which a race condition can form.

The distributed storage engine ensures this isolation works across a distributed cluster - across availability zones and regions - not just on a single node.

This is the difference between agents that happen to run in the same system and agents that genuinely *collaborate* through shared understanding.

4.3 Context engineering as composable queries

A well-engineered context assembly for a single agent decision might combine:

- User preferences from persistent file memory.
- Semantically relevant past conversations from Spectron's vector index.
- Structured entity facts from Spectron's knowledge graph with temporal metadata.
- Real-time state changes pushed via SurrealDB live queries.
- Domain knowledge from SurrealDB's schema-enforced ontological graph.

Because SurrealDB is multi-model, all of these signals compose in a single SurrealQL query - vector similarity, graph traversal, temporal filtering, and document lookup in one statement,

within one ACID transaction, returning one consistent view.

In a fragmented architecture, this is five API calls to five different systems with five different consistency models. The context assembly layer must reconcile conflicting timestamps, handle partial failures, and merge results with application-level logic. In SurrealDB, it is one query.

5. Why memory middleware is architecturally insufficient

5.1 The consistency boundary problem

Memory middleware operates above the database layer. It can enforce its own internal consistency - ensuring that its API returns coherent results - but it cannot enforce transactional consistency across the databases it orchestrates. When the vector store and the graph database have different views of the same entity, the middleware must choose which view to present. This choice is made at query time, based on timestamps or heuristics, and it is non-deterministic.

In SurrealDB, there is no choice to make. The graph edge and the vector embedding exist in the same transactional record. A query that traverses the graph and performs vector similarity over the same data is resolved from a single transactional snapshot. The consistency boundary is the ACID transaction itself - the strongest guarantee a database can provide.

5.2 The infrastructure gap

Memory middleware products do not own their storage layer. They depend on whatever vector database, graph database, and relational database the customer provisions. This means:

- **No compute-storage separation.** The middleware cannot scale compute independently of storage.
- **No scale-to-zero.** The underlying databases run continuously regardless of agent activity.
- **No instant branching.** Cloning the agent's memory environment requires copying data across multiple systems.
- **No native disaster recovery.** Each underlying system has its own backup and recovery procedures.
- **No unified cost model.** The customer pays for the middleware layer *and* for every underlying database separately.

The SurrealDB stack owns the full vertical - from object storage to agentic memory. The infrastructure properties of the storage layer (scale-to-zero, instant branching, 99.999999999% durability, cross-AZ cost reduction) flow directly through the database engine into the memory layer. A single system provides a single cost model, a single operational surface, and a single

consistency boundary.

5.3 Scope limitation

Memory middleware addresses one slice of what production agents need: remembering conversational context. Production agents also need:

- **Knowledge graphs with schema-enforced ontologies** - governed entity types with constrained relationships, not just extracted entities.
- **Application data** - user profiles, product catalogs, operational state, business rules in the same transactional system.
- **Real-time subscriptions** - live queries that push context changes instantly, not polling.
- **Authentication and access control** - row-level permissions enforced at the database layer.
- **API endpoints** - HTTP routes served by the database, not a separate API framework.
- **Event-driven workflows** - triggers and reactive logic without external message queues.

SurrealDB addresses all of these in a single system. Memory middleware addresses conversational recall and requires the organization to maintain separate infrastructure for everything else.

6. Why specialist agent databases are incomplete

Specialist agent databases - systems purpose-built for agent memory that combine graph and vector capabilities in a single product - correctly identify that the flat vector model is insufficient. Their graph-augmented retrieval approaches share the architectural intuition that drives Spectron: entities need types, relationships need structure, and temporal context matters.

However, combining graph and vector in one product is not the same as unifying them in one engine. Systems like HydraDB self-host a separate vector store alongside their graph, orchestrating between them at the application layer. This internalizes the multi-system architecture rather than eliminating it. They are incomplete in several dimensions:

6.1 No native document model

Agent memory is not the only data an agent needs. In production, the context layer must hold application data, business rules, user profiles, and operational state alongside memory. Specialist agent databases treat every piece of data as a node or an edge - they lack a native document model where a record can be a rich, nested, schema-validated JSON document with its own access control rules, event triggers, and computed fields.

In SurrealDB, a graph edge *is* a document. This is not a minor distinction - it means that relationships can carry the full richness of structured data without being reduced to key-value properties.

6.2 No ACID across all data models

Combining graph and vector capabilities in a single product is necessary but not sufficient. The critical question is whether graph operations and vector operations are part of the same ACID transaction. When a system self-hosts a separate vector store alongside its graph - as HydraDB does - the graph can update while the vector index is stale, or vice versa. A query that traverses the graph and performs vector similarity may see inconsistent state - a graph edge that references a vector embedding that has been updated but not yet committed.

SurrealDB provides full ACID transactions across all data models. A single SurrealQL query that traverses a graph, performs vector similarity, filters by temporal range, and updates a document does so within a single transaction with snapshot isolation.

6.3 No enterprise governance

Production agent systems in enterprise environments require governance primitives that specialist databases do not provide:

- **Row-level permissions** - Different agents, users, or tenants see different subsets of the data.
- **Field-level security** - Sensitive fields (PII, financial data) are redacted based on the caller's access level.
- **Schema enforcement** - The ontology is not just a convention; it is enforced at write time.
- **Audit trails** - Every query, every mutation, every access decision is traceable and reproducible.

SurrealDB provides `DEFINE ACCESS` with declarative permissions in the schema. Access control is enforced at the database layer - not at the application layer, not at the middleware layer. For regulated industries deploying production agent systems, this is non-negotiable.

6.4 No cloud-native storage

Specialist agent databases couple compute to storage. Scaling requires provisioning more nodes with more disk. Disaster recovery is an operational procedure - backup to external storage, restore from backup. Idle environments consume resources continuously.

The SurrealDB distributed storage engine provides compute-storage separation, object-storage backing, scale-to-zero, instant branching, and native disaster recovery. These are architectural

properties, not operational procedures. An agent's memory environment can be branched in seconds for testing. A failed availability zone is recovered by pointing new nodes at the same object-storage bucket. An idle deployment costs nothing.

6.5 Single-product MCP vs. ecosystem MCP

Several specialist agent databases - notably HelixDB - highlight MCP (Model Context Protocol) support as a differentiator. This is directionally correct: MCP is becoming the standard protocol for agent-to-tool connectivity, and native MCP support removes the need for custom SDK integration.

However, single-product MCP support connects an agent to *one tool*. The SurrealDB ecosystem provides MCP servers across the entire platform: SurrealDB's MCP server for structured data queries and schema management, Spectron's MCP server for memory ingestion and knowledge graph retrieval, and SurrealDB Cloud's MCP server for infrastructure provisioning and monitoring. An agent connected to the Surreal ecosystem can query data, store and retrieve memories, and manage infrastructure - all through standard MCP tool calls, all discoverable through the same protocol.

This is the difference between connecting an agent to a database and connecting an agent to a platform.

7. Discussion

7.1 The vertical integration advantage

The SurrealDB agent infrastructure stack is the only system that owns the complete vertical from object storage to agentic memory. This vertical integration is not a product strategy - it is an architectural requirement. The properties that production agent systems need - ACID consistency across data models, compute-storage separation for scale economics, instant branching for memory environments, and real-time subscriptions for context delivery - can only be provided when all layers of the stack are co-designed.

A memory layer built on top of a database it does not control cannot offer scale-to-zero. A database that does not own its storage engine cannot offer instant branching. A storage engine without a multi-model database on top cannot offer composable retrieval. The capabilities emerge from the integration, not from any individual layer.

7.2 Evaluating agent memory

Agent memory quality is not a single metric. The LongMemEval benchmark [10] defines six cognitive capabilities that a production memory system must support:

Capability	What it tests
Information extraction	Recall explicit facts from large, noisy conversations
Preference extraction	Identify and retain implicit user preferences - not just stated facts, but incl
Multi-session reasoning	Combine facts distributed across temporally and contextually disconnected
Temporal reasoning	Preserve and reason over chronology - distinguish current state from histo
Knowledge updates	Overwrite outdated facts when user preferences or states change, without l
Abstention	Identify insufficient information and refuse to answer rather than hallucin

These categories provide a useful framework for evaluating any memory system. A system that excels at extraction but fails at temporal reasoning will produce stale answers. A system that handles updates but cannot abstain will hallucinate when evidence is missing. Production-grade memory requires consistent performance across all six dimensions.

However, conversational recall benchmarks - however well-designed - test only one surface of what production agent systems require. Production readiness requires evaluation across dimensions that no current benchmark tests:

- **Multi-model query consistency** - Can the system guarantee that a graph traversal and a vector similarity search return data from the same transactional snapshot?
- **Concurrent write coordination** - Do multi-agent writes maintain ACID isolation, or do they introduce race conditions?
- **Retrieval relevance at enterprise scale** - Does relevance hold with millions of entities and billions of relationships, or does it degrade with corpus size?
- **Infrastructure resilience** - What happens when a node fails? An availability zone goes down? How quickly does the system recover?
- **Governance compliance** - Can the system enforce row-level access control across graph, vector, and document operations in a single query?

The SurrealDB stack is designed to satisfy all of these requirements. Conversational recall is one dimension of a much larger surface area.

7.3 Memory lifecycle: what to remember, what to forget

Production memory systems must manage the full lifecycle of knowledge - not just ingestion and retrieval, but also decay and eviction. Infinite memory retention creates its own problems:

- **Retrieval noise.** As the memory corpus grows, obsolete and low-relevance facts accumulate, drowning out critical information in retrieval results.
- **Stale fact contamination.** Facts that were true months ago but have since been superseded can contaminate an agent's reasoning if they remain at the same retrieval priority as current facts.

- **Cost and latency.** Larger indexes are more expensive to store and slower to search.

Effective memory systems need retention policies: which memories are high-value (medical allergies, core preferences) and should never decay, which are transient (a coffee order, a one-time scheduling request) and can be evicted, and which fall in between and should be archived to lower-priority tiers.

Some systems approach this with probabilistic decay functions that automatically prune memories based on age and access frequency. This introduces a risk: probabilistic deletion can discard information that appears low-value but is contextually critical. A rarely accessed medical allergy is more important than a frequently accessed coffee preference - but a frequency-based decay function treats them inversely.

SurrealDB's approach is deterministic. Facts carry explicit temporal metadata (`created_at`, `invalid_at`). Queries filter by time range and validity. The application - or a governance policy defined in the schema - decides what is retained and what is evicted, based on business rules rather than probabilistic heuristics. Combined with the distributed storage engine's object-storage economics, the cost of retaining large memory corpora is structurally low, reducing the pressure to prune aggressively.

7.4 Atomic context

The central architectural claim of this paper is that context for AI agents must be **atomic** - a single memory is not a collection of pointers across different systems but a unified record within a single transactional boundary.

In SurrealDB, a graph edge is a document. It can hold the weight of a relationship, the timestamp of the last interaction, the vector embedding of the context, and arbitrary metadata - all retrievable in a single query. When Spectron extracts a fact from a conversation, the entity, the relationship, the embedding, and the temporal metadata all live in the same transactional record.

This is the property that fragmented architectures cannot replicate. No amount of orchestration middleware can make multiple databases behave as a single ACID-compliant transactional system. The consistency boundary stops at the middleware API. Below that boundary, each database has its own view of the world. Above that boundary, the middleware presents a reconciled view that is, at best, eventually consistent.

The context layer must be atomic because agents make decisions in real time based on the context they receive. If that context is inconsistent - if the graph says one thing and the vector store says another - the agent's reasoning is compromised. ACID transactions are the only mechanism that guarantees a consistent view across all data models at query time.

8. Conclusion

The context layer for AI agents is an infrastructure problem, not a middleware problem. It requires a purpose-built database that unifies documents, graphs, vectors, and time-series in a single ACID-compliant transactional engine - backed by a cloud-native storage layer that separates compute from storage, scales to zero, branches instantly, and recovers from failures by design.

The SurrealDB agent infrastructure stack provides this: a distributed ACID storage engine on cloud object storage, a multi-model database with built-in auth, APIs, and real-time subscriptions, a structured agentic memory layer with entity extraction, knowledge graphs, and temporal fact tracking, and a persistent file memory for agent working state.

Memory middleware hides fragmentation but does not eliminate it. Specialist agent databases solve one dimension but leave infrastructure, governance, and scope unaddressed. Analytical platforms optimize for the wrong workload. The context layer must own the full vertical - from storage to memory - because the properties that production agents need emerge from the integration of all layers, not from any individual one.

From object storage to structured agent memory. One stack. One transaction. Full context.

References

[1] a16z, "The Emerging Architecture for AI Agent Infrastructure," a16z Research, 2026.

[2] SurrealDB, Inc., "The Context Layer for AI Agents," Internal positioning document, 2026.

[3] P. Chhikara, D. Khant, S. Aryan, T. Singh, D. Yadav, "Mem0: Building Production-Ready AI Agents with Scalable Long-Term Memory," arXiv:2504.19413, 2025.

[4] S. Ratnaparkhi, N. Srivastava, A. Garg, P. Garg, T. Kumar, "Hydra DB: Beyond Context Windows for Long-Term Agentic Memory," HydraDB, 2026.

[5] D. Ford, "Introducing Contextual Retrieval," Anthropic Engineering Blog, 2024.

[6] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, P. Liang, "Lost in the Middle: How Language Models Use Long Contexts," arXiv:2307.03172, 2023.

[7] D. Edge, H. Trinh, N. Cheng, J. Bradley, A. Chao, A. Mody, S. Truitt, J. Larson, "From local to global: A graph RAG approach to query-focused summarization," arXiv:2404.16130, 2024.

[8] Databricks, "Lakebase: The Database Built on the Lakehouse," Databricks Blog, 2025.

[9] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, D. R. K. Ports, "Building Consistent Transactions with Inconsistent Replication," ACM SOSP, 2015.

[10] D. Wu, H. Wang, W. Yu, Y. Zhang, K.-W. Chang, D. Yu, "LongMemEval: Benchmarking Chat Assistants on Long-Term Interactive Memory," ICLR, 2025.

[11] A. Maharana, D.-H. Lee, S. Tulyakov, M. Bansal, F. Barbieri, Y. Fang, "Evaluating very long-term conversational memory of LLM agents," arXiv:2402.17753, 2024.