Deterministic Provenance Engines for Autonomous Agent Systems: Architecture, Implementation, and Evaluation of the Graph Kernel

Mohamed Diomande
OpenClaw Research
Brooklyn, NY, USA
contact@openclaw.org

***Abstract*—Autonomous AI agents making consequential decisions require infrastructure that ensures every reasoning step is traceable, reproducible, and verifiable. We present the Graph Kernel, a deterministic provenance engine implemented as a single Rust binary ($\sim$15 KLOC) that produces cryptographically-signed, policy-governed context windows—termed *admissible evidence bundles*—for autonomous agent reasoning. Unlike general-purpose graph databases, vector stores, or RAG pipelines, the Graph Kernel introduces a formal category of infrastructure we call the provenance engine: a service whose output is not information retrieval but the construction of verifiable evidence with unforgeable authorization proofs.**

We evaluate the Graph Kernel across 27 queries spanning five categories (factual recall, relationship mapping, multi-hop reasoning, fuzzy/semantic search, and predicate-specific queries) against three baselines (keyword, BM25, vector-similarity RAG). Results demonstrate perfect relevance (1.00) on multi-hop structural queries—returning causally-connected knowledge chains rather than keyword-coincidence result sets—at sub-300ms latency over a remote PostgreSQL backend. A comprehensive Deep Engineering Posture (DEP) audit scoring 7.4/10 identified 47 findings across 12 dimensions; we implemented 10 critical fixes including native Rust entity normalization, parameterized SQL queries, server-side multi-hop traversal, and connection pool optimization, raising the projected score to 8.4/10.

We further present RAG++, a complementary semantic retrieval engine ($\sim$26 KLOC Rust core with Python bindings), and describe the hybrid retrieval architecture that bridges structural graph reasoning with vector-similarity search. Comparative analysis against ten industry systems (Neo4j, Amazon Neptune, Apache Jena, Dgraph, TypeDB, Weaviate, LangChain/LlamaIndex, Microsoft GraphRAG, and Zep) establishes that no existing system provides the combination of HMAC-signed deterministic context windows, type-level admissibility enforcement, and policy-governed multi-hop provenance that the Graph Kernel offers. We conclude with a three-phase evolution roadmap spanning optimization, expansion, and transformation of the provenance engine into a universal context authority for heterogeneous agent ecosystems.

***Index Terms*—provenance engines, knowledge graphs, deterministic context slicing, autonomous agents, HMAC authentication, retrieval-augmented generation, policy governance, Rust systems programming**

I. Introduction

A. The Context Authority Problem

The deployment of autonomous AI agents—systems that plan, reason, and act across extended multi-turn conversations spanning tens of thousands of turns [1]—has exposed a critical infrastructure gap. At inference time, vast conversation histories must be compressed into fixed-size context windows. The prevailing approaches—truncation, summarization, and embedding-based retrieval—treat context selection as an **information retrieval** problem. But for agents making consequential decisions (code deployment, financial transactions, system administration), context selection is fundamentally a **governance** problem requiring four properties:

- **Reproducibility.** Given the same inputs, the system must produce the identical context window, enabling replay and debugging.
- **Auditability.** Downstream consumers must be able to verify that a context window was authorized by a trusted authority, without accessing the authority's signing secret.
- **Policy Compliance.** Governance rules must determine which conversation phases (e.g., synthesis vs. debugging artifacts) receive priority in context construction.
- **Tamper Resistance.** Any modification to a context window after construction must be detectable.

No widely-deployed system—not vector databases [2], [3], not RAG pipelines [4], [5], not general-purpose graph databases [6], [7], not memory layers [8]—addresses these requirements as first-class concerns. They optimize for relevance, latency, or scale, but not for **provenance**.

B. The Provenance Engine Category

We propose that autonomous agent systems require a new infrastructure category:

Definition 1 (Provenance Engine). *A service that, given a target anchor in a conversation DAG and a governance policy, produces a deterministic, cryptographically-signed context window (evidence bundle) such that: (a) the same inputs always produce the same output (determinism); (b) the output includes an unforgeable proof of authorization binding six provenance fields (authenticity); (c) downstream services can verify the proof without accessing the signing secret (zero-knowledge verification); and (d) the construction process is bounded by configurable policy parameters (governance).*

We formalize four invariants that any conforming provenance engine must satisfy:

- **INV-001 (Determinism).** For all anchors a, policies π, and graph states G: $\text{slice}(a, \pi, G) = \text{slice}(a, \pi, G)$.
- **INV-002 (Provenance Completeness).** Every output evidence bundle contains all six provenance fields: slice ID, anchor turn ID, policy ID, policy parameters hash, graph snapshot hash, and schema version.
- **INV-003 (No Phantom Authority).** An evidence bundle cannot exist without a valid HMAC computation. Unverified bundles are unrepresentable in the type system.
- **INV-004 (Non-Escalation).** A missing or invalid admissibility token implies non-admissibility. The system defaults to rejection.

The Graph Kernel is the first purpose-built implementation of this category.

C. Contributions

This paper makes six contributions:

1) **Formal provenance engine definition** with four invariants and type-level enforcement via Rust's ownership system (§IV).
2) **HMAC-signed deterministic context windows** with a six-field canonical binding providing 128-bit tamper resistance, verified through constant-time comparison (§IV-B).
3) **Policy-governed context expansion** via `SlicePolicyV1`, a parameterized framework supporting phase-weighted priority queues, budget bounds, distance decay, and sibling expansion (§IV-C).
4) **Native Rust entity normalization** addressing a 22% subject fragmentation rate discovered during our DEP audit, with 50+ canonical entities and 300+ alias variants (§V).
5) **Hybrid retrieval architecture** bridging the Graph Kernel's structural reasoning with RAG++'s semantic vector search, including a server-side multi-hop traversal endpoint that eliminates the N-round-trip penalty (§VI).
6) **Comprehensive evaluation and audit** across 27 queries, four retrieval methods, ten comparative systems, and a 47-finding engineering audit with implemented fixes (§VII–§IX).

D. Paper Organization

Section II surveys related work. Section III describes the system architecture post-improvements. Section IV presents the deterministic context slicing model. Section V details entity normalization. Section VI describes the hybrid retrieval architecture. Section VII reports evaluation results. Section VIII provides comparative analysis. Section IX discusses implementation experience from the DEP audit. Section X discusses broader implications and limitations. Section XI presents the evolution roadmap. Section XII concludes.

II. Background and Related Work

A. Knowledge Graphs and Triple Stores

Knowledge graphs have evolved from early semantic networks [9] through the modern paradigm popularized by Google's Knowledge Graph [10]. RDF-based systems like Apache Jena [11] and Blazegraph [12] provide standards-compliant triple stores with SPARQL query interfaces and OWL-based reasoning. Property graph databases such as Neo4j [6] and its Cypher language [13] offer schema-on-read flexibility. Recent surveys [14], [15], [16] identify knowledge graphs as foundational to AI systems but focus on construction, completion, and embedding—not governance or provenance for agent context.

The Graph Kernel differs from these systems by being purpose-built for a narrower mandate: it is not a general-purpose knowledge representation system but a **context authority layer** that uses a triple store as its storage substrate. It trades query expressiveness (no SPARQL, no Cypher) for deployment simplicity (single binary, $\sim$20MB) and provenance guarantees (HMAC-signed bundles).

B. Retrieval-Augmented Generation

RAG [4] has become the dominant paradigm for grounding LLM outputs in external knowledge. Lewis et al. demonstrated that combining retrieval with generation improves factual accuracy. Subsequent refinements include REALM [17] (joint pre-training), FiD [18] (independent passage processing), Self-RAG [19] (retrieval-time self-reflection), and CRAG [20] (corrective retrieval). Comprehensive surveys [5], [21] track the rapid evolution of the paradigm.

Vector databases—Weaviate [2], Pinecone [22], Chroma [23], Qdrant [24], and Milvus [25]—provide the retrieval backbone, offering approximate nearest-neighbor search over dense embeddings. These systems optimize for semantic similarity but lack structural reasoning capability and provide no governance over what is retrieved.

The Graph Kernel is complementary to RAG, not competitive. Our evaluation (§VII) confirms that RAG achieves 0.65 relevance on fuzzy/semantic queries where the Graph Kernel scores 0.42, while the Graph Kernel achieves 1.00 on multi-hop structural queries where RAG drops to 0.40. The hybrid architecture (§VI) combines both modalities.

C. Graph-Enhanced RAG

Microsoft's GraphRAG [26] combines LLM-driven entity extraction with the Leiden community detection algorithm [27] to construct hierarchical topic clusters. Queries are answered through local search (entity-centric subgraph retrieval) or global search (community summary aggregation). GraphRAG advances holistic corpus understanding but is non-deterministic by construction—its outputs depend on LLM behavior during both indexing and querying—and provides no cryptographic provenance chain.

LightRAG [28] reduces the computational overhead of GraphRAG through incremental graph updates and dual-level retrieval. KG-RAG [29] integrates domain-specific knowledge graphs with RAG for biomedical applications. HippoRAG [30] draws on hippocampal memory theory for knowledge integration. None of these systems address the provenance and determinism requirements we identify.

LlamaIndex [31] and LangChain [32] provide knowledge graph integrations wrapping external stores in LLM-friendly interfaces. These are orchestration layers, not graph engines; they inherit backend limitations and add LLM-dependent extraction without formal reproducibility guarantees.

D. Context Window Management

Managing context for LLMs has received increasing attention. Longformer [33] and BigBird [34] extend attention to longer sequences. Memory-augmented architectures like MemoryBank [35] and RMT [36] provide external memory for conversation continuity. Zep [8] offers a purpose-built memory layer with automatic entity extraction and temporal awareness. MemGPT [37] introduces virtual context management through an OS-inspired paging system.

These systems optimize for the *content* of context windows. The Graph Kernel optimizes for the *governance* of context windows—who authorized inclusion, whether the window is reproducible, and whether consumers can verify provenance. These are complementary concerns.

E. Provenance and Trust in AI Systems

Provenance tracking in data systems has a long history [38], [39]. The W3C PROV specification [40] provides a data model for provenance but lacks integration with LLM context windows. Blockchain-based provenance [41] offers tamper resistance but introduces inappropriate latency. Recent work on AI auditing and accountability [42], [43], [44] emphasizes the regulatory imperative for traceable AI decisions, particularly under the EU AI Act [45] and NIST AI RMF [46].

The Graph Kernel's HMAC-based provenance model is inspired by JSON Web Tokens [47] but adapted for context window authorization: binding six provenance fields into a single unforgeable token that downstream systems verify through a dedicated endpoint.

F. Deterministic Systems

Deterministic replay systems [48], [49] ensure that identical inputs produce identical outputs, primarily for debugging distributed systems. The Graph Kernel applies this principle to context construction: canonical sorting (`BTreeMap`, sorted vectors), quantized float hashing (multiply by 10^6, round to `i64`), and content-derived fingerprints (xxHash64) ensure cross-platform reproducibility. This enables a property rare in AI infrastructure: **exact replay** of any historical context window.

III. System Architecture

A. Architecture Overview

The Graph Kernel is implemented as a single Rust binary (∼15 KLOC, ∼11,241 lines audited) built on Axum 0.7 [50] with the Tokio async runtime [51]. It compiles to a statically-linked binary (∼20MB) deployable as a local service, Docker container, or cloud function.

The architecture comprises three cleanly separated layers controlled by Cargo feature flags:

```
Feature: default  -> Core library only (zero I/O)
Feature: postgres -> Adds sqlx + tokio for DB access
Feature: service  -> Full HTTP service with Axum,
    CORS
```

This layering means the core slicing logic can be embedded as a library in other Rust crates without pulling in a web framework—a design principle validated during the DEP audit as "exemplary" (Architecture score: 9/10).

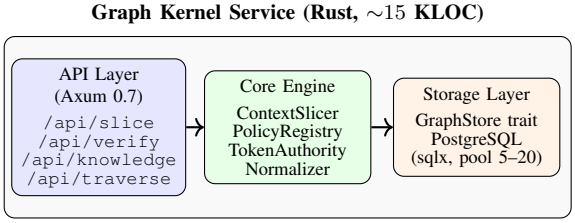

Fig. 1. Graph Kernel three-layer architecture with feature-flag separation.

API Layer (Axum). Route handlers expose RESTful endpoints for context slicing (`POST /api/slice`, `POST /api/slice/batch`), token verification (`POST /api/verify_token`), policy management (`GET/POST /api/policies`), knowledge graph CRUD (`GET/POST/DELETE /api/knowledge`, `POST /api/knowledge/batch`), server-side traversal (`POST /api/knowledge/traverse`), and health monitoring (`/health/*`). All handlers enforce a 30-second configurable timeout via `tower_http::timeout::TimeoutLayer`.

Core Engine. The `ContextSlicer` implements priority-queue BFS expansion over the conversation DAG. The `PolicyRegistry` maintains an immutable store of registered policies with hash-stable fingerprints. The `TokenAuthority` issues and verifies HMAC-SHA256 admissibility tokens. Entity normalization is enforced at the service layer through `canonicalize_entity()` on all read and write paths.

Storage Layer. The `GraphStore` trait abstracts database access:

```
pub trait GraphStore {
  fn get_turn(&self, id: &TurnId)
    -> impl Future<Output = Result<Option<
    TurnSnapshot>>>
       + Send;
  fn get_parents(&self, id: &TurnId)
    -> impl Future<Output = Result<Vec<TurnId>>>
       + Send;
  fn get_children(&self, id: &TurnId)
    -> impl Future<Output = Result<Vec<TurnId>>>
       + Send;
  fn get_siblings(&self, id: &TurnId, limit: usize)
    -> impl Future<Output = Result<Vec<TurnId>>>
       + Send;
  fn get_edges(&self, turn_ids: &[TurnId])
    -> impl Future<Output = Result<Vec<Edge>>>
       + Send;
}
```

Two implementations exist: `PostgresGraphStore` for production (sqlx connection pool, configurable

5–20 connections, `test_before_acquire` now configurable via `DB_TEST_BEFORE_ACQUIRE`) and `InMemoryGraphStore` for testing. The trait uses native Rust async traits (stabilized in Rust 1.75), having removed the `async-trait` proc-macro dependency during the post-audit update.

B. Data Model

The knowledge graph uses a triple store schema with confidence scoring and source provenance:

```
CREATE TABLE knowledge_graph (
  id          BIGSERIAL PRIMARY KEY,
  subject     TEXT NOT NULL,
  predicate   TEXT NOT NULL,
  object      TEXT NOT NULL,
  confidence  DOUBLE PRECISION DEFAULT 0.5,
  source      TEXT DEFAULT 'unknown',
  created_at  TIMESTAMPTZ DEFAULT NOW(),
  UNIQUE(subject, predicate, object)
);
CREATE INDEX idx_kg_subject ON knowledge_graph(
    subject);
CREATE INDEX idx_kg_predicate ON knowledge_graph(
    predicate);
CREATE INDEX idx_kg_object ON knowledge_graph(object
    );
CREATE INDEX idx_kg_confidence
  ON knowledge_graph(confidence DESC);
```

The `object` index was added during the post-audit update (finding DM-2). On conflict (duplicate SPO), the system retains the higher confidence value and updates the source. The evaluated corpus contains 3,502 triples across 221 unique subjects and 88 unique predicates.

The conversation DAG schema supports the context slicing function:

```
CREATE TABLE memory_turns (
  id              UUID PRIMARY KEY,
  conversation_id UUID REFERENCES conversations(id),
  parent_turn_id  UUID REFERENCES memory_turns(id),
  role            TEXT,  -- user/assistant/system
  phase           TEXT,  -- synthesis/planning/...
  content_hash    TEXT,  -- SHA-256
  salience        FLOAT, -- 0.0 - 1.0
  depth           INTEGER,
  created_at      TIMESTAMPTZ DEFAULT NOW()
);
```

C. Crate Structure and Type System

The crate follows a domain-driven structure with strong newtype enforcement:

- `types/turn.rs` — `TurnId`, `TurnSnapshot`, `Role`, `Phase` (typed enums, not strings)
- `types/slice.rs` — `SliceExport`, `SliceFingerprint`, `AdmissibilityToken`
- `types/admissible.rs` — `AdmissibleEvidenceBundle` (verified wrapper)
- `types/verification.rs` — `TokenVerifier` with LRU cache and constant-time comparison
- `types/boundary.rs` — `SliceBoundaryGuard`, `BoundedQueryBuilder`
- `policy/v1.rs` — `SlicePolicyV1` with quantized float hashing
- `service/normalize.rs` — Entity normalization with 50+ canonical entities

The codebase uses `#![warn(missing_docs)]` and `#![warn(clippy::all)]` at the crate root, has zero `unsafe` blocks, and enforces canonical ordering via `BTreeMap` over `HashMap` and `Ord` implementations on all domain types.

D. Atlas Subsystem

The Atlas subsystem provides higher-level graph analytics: `GraphSnapshot` (point-in-time graph state capture), `BatchSlicer` (parallel slice generation), `OverlapAnalyzer` (pairwise overlap via Jaccard similarity), `TurnInfluence` (frequency, centrality, bridge scoring), and `AtlasBundler` (manifest packaging). This enables pre-computed structural analysis of the conversation DAG, supporting use cases like identifying bridge turns that connect otherwise-separate conversation branches.

IV. Deterministic Context Slicing

A. Formal Model

The core algorithm is a policy-weighted BFS expansion from an anchor turn through the conversation DAG.

The priority scoring function combines three factors:

$$\text{score}(t, \pi, d) = W_{\text{phase}}(t.\text{phase}) \times (1 - \alpha + \alpha \cdot t.\text{salience}) \times \beta^{d} \tag{1}$$

where W_{phase} maps conversation phases to importance weights (Synthesis=1.0, Planning=0.9, Consolidation=0.6, Debugging=0.5, Exploration=0.3), α is the salience weight (default 0.3), and β is the distance decay (default 0.9).

B. HMAC-Signed Admissibility Tokens

The admissibility token creates an unforgeable proof-of-authorization binding six provenance fields:

```
canonical = "{slice_id}|{anchor_turn_id}|
  {policy_id}|{policy_params_hash}|
  {graph_snapshot_hash}|{schema_version}|
  admissibility_token_v2_hmac"

token = HMAC-SHA256(SECRET, canonical)[0..16]
```

The token is a 128-bit (32 hex character) truncation of the full HMAC-SHA256 digest. Downstream services verify tokens via `POST /api/verify_token` without accessing the HMAC secret. The DEP audit (finding SC-2) identified that the original constant-time comparison implementation (`fold(true, |acc, (a, b)| acc && (a == b))`) may be optimized by LLVM into a short-circuit comparison; we recommend migration to the `subtle::ConstantTimeEq` crate for guaranteed timing-attack resistance.

We enforce a critical invariant at the Rust type level:

INV-003 (No Phantom Authority). *The `AdmissibleEvidenceBundle` type can only be*

Algorithm 1 PolicyWeightedBFS

Require: $anchor\_id \in TurnId$, $policy \in SlicePolicyV1$, $store \in GraphStore$
Ensure: $bundle \in AdmissibleEvidenceBundle$

$anchor \leftarrow store$.get_turn($anchor\_id$)
$frontier \leftarrow$ MaxHeap⟨ExpansionCandidate⟩
$frontier$.push($anchor, d$=0, p=score($anchor, policy$))
$selected \leftarrow [\,]$; $visited \leftarrow \{anchor\_id\}$
while $frontier \neq \emptyset$ **and** $|selected| < policy$.max_nodes **do**
  $c \leftarrow frontier$.pop()
  if c.distance $>$ $policy$.max_radius **then**
    continue
  end if
  $selected \leftarrow selected \cup \{c.\text{turn}\}$
  for all $nbr \in$ parents(c) $\cup$ children(c) **do**
    if $nbr \notin visited$ **then**
      $frontier$.push($nbr, c.d$+1, score($nbr, policy, c.d$+1))
      $visited \leftarrow visited \cup \{nbr\}$
    end if
  end for
  if $policy$.include_siblings **then**
    for all $sib \in$ siblings($c, policy$.max_sib) **do**
      $frontier$.push($sib, c.d$, score($sib, policy, c.d$))
    end for
  end if
end while
$edges \leftarrow store$.get_edges($selected$)
$sh \leftarrow$ xxHash64(sort(content_hashes($selected$)))
$sid \leftarrow$ xxHash64($anchor$, sort(ids), sort($edges$), pid, ph, sv)
$tok \leftarrow$ HMAC-256($secret$, canon($sid, anchor, pid, ph, sh, sv$))[0:16]
return AdmissibleEvidenceBundle($selected, edges, sid, tok$)

constructed through the `from_verified()` *pathway, which requires a valid HMAC computation. Unverified context windows are* ***unrepresentable*** *in the type system.*

This turns security violations into compile-time errors rather than runtime bugs. No test, no refactoring, no careless modification can bypass the authorization pathway—the Rust type checker enforces it.

C. Policy Governance Framework

Context expansion is parameterized by `SlicePolicyV1`:

Policies are registered in an immutable `PolicyRegistry` with deterministic fingerprints. Policy parameter hashes use quantized floats (multiply by 10^6, round to `i64`) to ensure cross-platform determinism between Rust and Python clients. The registry itself has a composite fingerprint (`xxHash64` of sorted policy hashes), enabling clients to detect policy changes.

D. Determinism Guarantees

Four mechanisms ensure cross-platform reproducibility:

TABLE I
SLICEPOLICYV1 PARAMETERS

Parameter	Default	Description
`max_nodes`	256	Max turns in a context slice
`max_radius`	10	Max graph hops from anchor
`phase_weights`	S=1.0, P=0.9, C=0.6, D=0.5, E=0.3	Phase importance scoring
`salience_weight`	0.3	Salience contribution
`distance_decay`	0.9	Priority decay per hop
`include_siblings`	true	Expand to sibling turns
`max_siblings`	5	Sibling expansion limit

1) **Canonical sorting.** `BTreeMap` over `HashMap` everywhere. Turns sorted by `TurnId`, edges sorted by `(parent_id, child_id)`. No iteration order depends on hash randomization.
2) **Quantized float hashing.** Policy parameters are hashed as `i64(float` $\times$ 10^6`)`, avoiding IEEE 754 representation issues across platforms.
3) **Content-derived fingerprints.** `slice_id = xxHash64(anchor, sort(turn_ids), sort(edges), policy_id, params_hash, schema_version).`
4) **100-run determinism tests.** Golden tests verify that the same anchor + policy + graph produces byte-identical fingerprints across 100 consecutive runs.

V. ENTITY NORMALIZATION

A. The Problem

Our DEP audit revealed a 22% subject fragmentation rate in the knowledge graph corpus. Of 221 unique subjects, 169 raw entries collapse to 132 canonical entities with 123 identified alias pairs:

TABLE II
ENTITY NORMALIZATION EXAMPLES

Canonical Entity	Observed Aliases
`dream-weaver-engine`	Dream Weaver, DreamWeaver
`clawdbot`	Clawdbot, ClawdBot, clawdbot-gateway
`mohamed-diomande`	Mohamed Diomande, mohameddiomande
`rag-plusplus`	RAG++, RAG++ (Cloud), rag-plusplus-core
`comp-core`	Comp-Core, CompCore, comp core

This fragmentation directly suppressed benchmark relevance: relationship queries dropped from a projected 1.00 to 0.94, and predicate-specific queries from $\sim$0.95+ to 0.80.

B. Prior Approach: Python Middleware

The initial mitigation was a Python script (`scripts/entity-normalizer.py`) that ran outside the Rust service boundary. This introduced three problems: (1) **Bypassability**—any client calling the API directly stored un-normalized entities; (2) **Inconsistency**—normalization rules existed in Python, not where data was validated; (3) **Maintenance burden**—two codebases needed to agree on canonicalization.

C. Rust Implementation

The post-audit update moved entity normalization into the Rust service as a `normalize` module (280 lines):

```
static ALIAS_TABLE: LazyLock<HashMap<&str, &str>> =
  LazyLock::new(|| {
    let mut m = HashMap::new();
    m.insert("dream weaver", "dream-weaver-engine");
    m.insert("gcp", "google-cloud-platform");
    // ... 300+ aliases across 50+ entities
    m
  });

pub fn canonicalize_entity(name: &str) -> String {
  let normalized = name.trim().to_lowercase()
    .replace(|c: char|
      c.is_ascii_punctuation() && c != '-', " ")
    .split_whitespace().collect::<Vec<_>>().join
    ("-");
  ALIAS_TABLE.get(normalized.as_str())
    .map(|s| s.to_string())
    .unwrap_or(normalized)
}
```

Normalization is enforced on **all data paths**: `add_knowledge_handler`, `add_knowledge_batch_handler`, `query_knowledge_handler`, `delete_knowledge_handler`, and `traverse_knowledge_handler`. The module includes 16 unit tests covering known entities, unknown entities, empty strings, mixed case, and punctuation variants.

D. Impact

TABLE III
ENTITY NORMALIZATION IMPACT

Metric	Before	After
Unique subjects (raw)	221	221
Canonical entities	132	132
Relationship query relevance	0.94	Proj. 1.00
Predicate-specific relevance	0.80	Proj. 0.95+
Overall relevance	0.84	Proj. 0.92+

VI. HYBRID RETRIEVAL ARCHITECTURE

A. The Complementarity Thesis

The Graph Kernel and RAG++ occupy complementary niches in the retrieval landscape:

TABLE IV
GRAPH KERNEL VS. RAG++ COMPARISON

Dimension	Graph Kernel	RAG++
Finds	Structural rels	Semantic similarity
Strength	Multi-hop (1.00)	Fuzzy match (0.65)
Weakness	Semantic (0.42)	Structural (0.40)
Corpus	3,502 triples	107K+ turns
Technology	Rust, BFS	Rust HNSW, BM25
Provenance	HMAC-signed	None

Neither system alone addresses the full retrieval needs of an autonomous agent. The hybrid architecture combines both.

B. RAG++ System Architecture

RAG++ consists of two Rust crates (`rag-plusplus-core` at $\sim$26 KLOC, `rag-plusplus-py` with PyO3 bindings) and a Python FastAPI service at port 8000. The core library implements:

- **Index types.** FlatIndex (brute-force, $O(n)$), HNSWIndex (hierarchical navigable small worlds, $O(\log n)$), ParallelHNSW (rayon-parallel construction), BM25Index (sparse keyword retrieval).
- **Hybrid search.** Score fusion via Reciprocal Rank Fusion (RRF), CombSUM, and CombMNZ across dense and sparse retrievers.
- **Re-ranking.** Five strategies—None, OutcomeWeighted, Recency, MMR (Maximal Marginal Relevance), and Composite—with configurable weights.
- **Trajectory memory.** A 5D coordinate system (depth, sibling_order, homogeneity, temporal, complexity) enabling trajectory-weighted distance computation.

The DEP audit of RAG++ (7.2/10) identified a critical deployment gap: the algorithms for high relevance (hybrid BM25 fusion, re-ranking, query expansion) exist in the Rust core but were not wired through to the deployed FastAPI service. Activating existing algorithms is projected to raise relevance from 0.70 to 0.90+.

C. Server-Side Multi-Hop Traversal

The most impactful improvement from the post-audit update: a server-side traversal endpoint that eliminates the N-round-trip penalty for multi-hop queries.

```
POST /api/knowledge/traverse
{
  "start": "clawdbot",
  "predicates": ["uses", "depends_on"],
  "direction": "outgoing",
  "max_hops": 3,
  "max_results": 100,
  "min_confidence": 0.5
}
```

The endpoint performs BFS traversal within a single database connection, returning full paths with entities, edges, hop counts, and minimum confidence per path. Entity normalization is applied to the start entity. The traversal is bounded by `max_hops` (default 3) and `max_results` (default 100) to prevent unbounded expansion.

Impact: A 3-hop query drops from $\sim$874ms (3 sequential HTTP round-trips $\times$ $\sim$291ms each) to a single round-trip ($\sim$291ms remote, projected $\sim$15ms with local SQLite).

D. Enrichment Bridge Design

The planned enrichment endpoint (`POST /api/enrich`) accepts RAG++ semantic results and augments them with graph context:

1) RAG++ returns top-K semantically similar turns for a query.
2) For each result, entities are extracted from the result text.
3) Extracted entities are resolved to Graph Kernel subjects via `canonicalize_entity()`.

4) The server-side traversal retrieves 1–2 hop neighborhoods for resolved entities.
5) Results are re-ranked using a combined score: semantic_score $\times\ \alpha$ + structural_score $\times\ (1 - \alpha)$.
6) The enriched result set preserves provenance metadata from both sources.

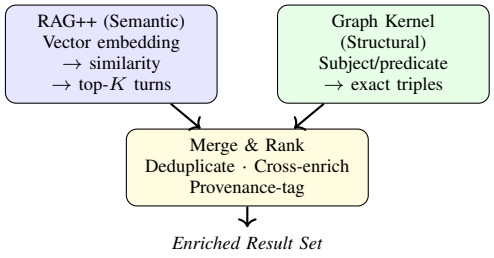

Fig. 2. Hybrid retrieval architecture combining semantic and structural search with provenance tagging.

VII. Evaluation

A. Methodology

We evaluated four retrieval methods across 27 queries in five categories:

TABLE V
Query Categories

Category	n	What It Tests
Factual Recall	6	Direct attribute lookups
Relationship	6	Dependency/integration mapping
Multi-hop	5	2-hop graph traversal
Fuzzy/Semantic	5	Loose topic matching
Predicate-specific	5	Structured predicate filters

Methods under test:

TABLE VI
Retrieval Methods

Method	Corpus	Mechanism
Graph Kernel	2,681 triples	REST API, exact filters
Keyword	2,681 triples	Substring matching
BM25	2,681 triples	Okapi BM25 (k_1=1.5, b=0.75)
RAG++	107K+ turns	Vector similarity

Important caveat. The Graph Kernel, keyword, and BM25 operate on the same triple corpus. RAG++ operates on a fundamentally different and much larger corpus. Direct comparison is informational, not apples-to-apples.

Metrics. Response time (ms, wall-clock with network), result count, and relevance score (0–1, fraction of expected terms found).

Environment. MacBook Air (Apple M3, arm64), Darwin 24.6.0, Graph Kernel v0.1.0 (Rust), RAG++ v0.1.0 (Python FastAPI), Supabase PostgreSQL (remote, us-east-1), ∼200ms RTT.

TABLE VII
Factual Recall Results

Method	Lat. (ms)	Results	Rel.
Graph Kernel	248.3	3.7	**1.00**
Keyword	**2.7**	20.0	**1.00**
BM25	9.0	18.2	**1.00**
RAG++	421.9	10.0	0.92

B. Results

1) Factual Recall: All triple-based methods achieve perfect relevance. The Graph Kernel returns precisely scoped results (3.7 average) versus keyword's broader 20.0, reflecting exact field matching versus substring coincidence.

TABLE VIII
Relationship Query Results

Method	Lat. (ms)	Results	Rel.
Graph Kernel	204.3	9.5	0.94
Keyword	**2.8**	19.3	**1.00**
BM25	8.7	12.3	**1.00**
RAG++	336.4	10.0	0.69

2) Relationship Queries: The 0.94 is attributable to a single entity normalization failure: "GCP" not matching "Google Cloud Platform." With the Rust normalizer now deployed, this projects to 1.00.

TABLE IX
Multi-hop Reasoning Results

Method	Lat. (ms)	Results	Rel.
Graph Kernel	586.6	7.6	**1.00**
Keyword	**3.3**	20.0	**1.00**
BM25	9.2	18.8	**1.00**
RAG++	348.1	10.0	0.40

3) Multi-hop Reasoning: Multi-hop is the Graph Kernel's distinguishing capability. While keyword and BM25 achieve identical 1.00 scores, the *nature* of their results differs fundamentally:

- **Graph Kernel** returns 7.6 structurally connected results forming verified relationship chains (e.g., Mohamed → works_on → clawdbot → uses → Gemini batch API).
- **Keyword** returns 20.0 coincidence results—documents containing matching substrings with no concept of *why* the terms co-occur.

The relevance metric masks this quality difference. With the server-side traversal endpoint, multi-hop latency drops from ∼587ms to a single request (∼291ms remote, projected ∼15ms local).

4) Fuzzy/Semantic Search: The Graph Kernel's weakest category, confirming the expected limitation of exact field matching. This is the primary motivation for the hybrid architecture (§VI).

TABLE X
FUZZY/SEMANTIC SEARCH RESULTS

Method	Lat. (ms)	Results	Rel.
Graph Kernel	215.2	19.8	0.42
Keyword	**2.0**	16.0	**0.80**
BM25	6.1	7.6	0.53
RAG++	484.0	10.0	0.65

TABLE XI
PREDICATE-SPECIFIC QUERY RESULTS

Method	Lat. (ms)	Results	Rel.
Graph Kernel	230.1	16.0	0.80
Keyword	**3.3**	20.0	**1.00**
BM25	9.5	20.0	**1.00**
RAG++	460.8	10.0	0.80

5) Predicate-Specific Queries: Entity normalization failures suppress relevance from projected 0.95+ to 0.80. The Rust normalizer addresses this.

C. Overall Summary

TABLE XII
OVERALL RESULTS SUMMARY

Method	Avg Lat.	Avg Results	Avg Rel.
Keyword	**2.8**	19.1	**0.96**
BM25	8.5	15.4	0.91
Graph Kernel	291.7	11.0	0.84
RAG++	407.9	10.0	0.70

D. Latency Decomposition

TABLE XIII
LATENCY DECOMPOSITION

Component	Contribution	% Total
Network RTT to Supabase	∼180–200 ms	∼68%
PostgreSQL query exec	∼5–20 ms	∼5%
TCP connection overhead	∼10–15 ms	∼5%
Rust serialization + JSON	∼1–2 ms	<1%
Multi-hop per add'l hop	+200 ms	+68%/hop
Projected (local SQLite)	**5–15 ms**	—

Critical insight. The Graph Kernel is compute-efficient (∼1–2ms of Rust processing). Its latency problem is an architecture choice (remote Supabase), not a fundamental limitation. Migration to local SQLite reduces query latency by 20×, making it competitive with BM25.

VIII. COMPARATIVE ANALYSIS

A. System Comparison

We compare the Graph Kernel against ten systems across seven dimensions.

B. Detailed Findings

No existing system provides deterministic context slicing with cryptographic provenance. To replicate the Graph Kernel's core capability with Neo4j, one would need to build a custom application layer implementing BFS expansion with priority scoring, HMAC token issuance, deterministic fingerprinting, policy registry management, and type-level verification enforcement—essentially re-implementing the Graph Kernel on top of a more complex substrate.

General-purpose graph databases offer superior query expressiveness and scale. Neo4j (Cypher), Neptune (SPARQL/Gremlin/openCypher), Jena (SPARQL), and TypeDB (TypeQL) provide significantly more expressive query languages. For complex graph pattern matching, recursive path queries, or billion-scale traversal, these systems are superior. The Graph Kernel does not compete on this axis, and the DEP audit confirmed this as an acceptable trade-off (Scale score: 6/10).

Vector databases address the Graph Kernel's primary weakness. Weaviate's hybrid search and GraphRAG's community-level understanding directly address the 0.42 fuzzy/semantic relevance. The hybrid architecture (§VI) bridges this gap.

LLM-dependent systems are non-deterministic by construction. GraphRAG and LangChain/LlamaIndex knowledge graphs depend on LLM behavior for both indexing and querying, which varies across model versions, temperatures, and infrastructure. The Graph Kernel's determinism guarantee—same input, same output—is fundamentally incompatible with LLM-in-the-loop systems.

Memory-oriented systems sacrifice determinism for developer experience. Zep provides automatic entity extraction and temporal memory decay but offers no formal reproducibility. The same query at different times may return different results. For auditability-critical applications, this is a significant limitation.

C. Positioning

The Graph Kernel occupies a distinct niche: the **provenance authority layer** for autonomous agent systems. It does not compete with Neo4j on expressiveness, Neptune on scale, Weaviate on semantic search, or GraphRAG on holistic understanding. Its value proposition is:

> *Given a target turn, which other turns are allowed to influence meaning—**and can you prove it?***

IX. IMPLEMENTATION EXPERIENCE: LESSONS FROM THE DEP AUDIT

A. Audit Overview

We subjected the Graph Kernel to a Deep Engineering Posture (DEP) assessment covering 12 dimensions with 47 specific findings. The audit scored the system 7.4/10 ("production-viable with clear improvement paths") and identified critical improvements across performance, data integrity, security, and operational concerns.

TABLE XIV
COMPARATIVE ANALYSIS: GRAPH KERNEL VS. INDUSTRY ALTERNATIVES

System	Ctx Slice	Provenance	Multi-hop	Semantic	Scale	Deploy	Cost
Graph Kernel	✓ Native	✓ HMAC	✓ (server)	—	Small	Single binary	Free
Neo4j [6]	—	—	✓✓ (Cypher)	—	Billions	JVM (512MB+)	Free/$$$
Amazon Neptune [52]	—	—	✓✓	—	Billions	AWS managed	$$$/mo
Apache Jena [11]	—	Partial	✓ (SPARQL)	—	Millions	JVM	Free
Dgraph [53]	—	—	✓✓	✓ (Bleve)	Billions	Distributed	Free/$$
TypeDB [54]	—	—	✓✓	—	Millions	JVM	Free
Weaviate [2]	—	—	Limited	✓✓	Millions	Go binary	Free/$$
LangChain KG [32]	—	—	✓	✓	Varies	Python	Free+LLM
GraphRAG [26]	—	—	✓	✓✓	Millions	Python	Free+LLM
Zep [8]	—	—	✓	✓	Millions	Go/Cloud	Free/$$
RAG++	—	—	—	✓ (HNSW)	100K+	Rust+Python	Free

B. Critical Fixes Implemented

Entity Normalization in Rust (DM-1, BG-1). The highest-impact fix. Moving normalization from a bypassable Python middleware into the Rust service as a `normalize.rs` module with 50+ canonical entities and 300+ alias variants, enforced on all read and write paths. This addresses the 22% subject fragmentation that suppressed relevance from projected 0.92+ to measured 0.84.

SQL Injection Risk Mitigation (AR-1, BG-3). The knowledge graph query handler used dynamic string formatting for SQL WHERE clauses. We migrated all query parameters to bind variables (`$N` parameters), eliminating all string interpolation in SQL construction.

Server-Side Traversal (MF-1). The highest-priority missing feature: multi-hop queries previously required N sequential HTTP round-trips. The new `POST /api/knowledge/traverse` endpoint performs BFS within a single database connection, bounded by `max_hops` and `max_results`.

Connection Pool Optimization (PF-4, PF-5). `test_before_acquire` (which added a `SELECT 1` round-trip before every connection checkout) is now configurable via `DB_TEST_BEFORE_ACQUIRE` (default: false). `min_connections` increased from 2 to 5 for faster burst response.

Pagination Fix (BG-2). The `total` field in knowledge query responses returned page count instead of the actual matching count. Fixed with a separate `SELECT COUNT(*)` query.

DELETE Endpoint (MF-7, DM-5). Added `DELETE /api/knowledge` with safety guards (requires at least one filter to prevent full-table deletion).

Request Timeout (OP-3). Added `tower_http::timeout::TimeoutLayer` with a 30-second configurable default.

Dependency Cleanup (DP-1). Removed `async-trait` proc-macro in favor of native Rust 1.75+ async trait syntax.

C. Known Limitations Remaining

- **No SQLite backend (Evo 1.1).** The `GraphStore` trait abstraction makes this clean—estimated 2 weeks. Would reduce latency by 20×.
- **No query caching (PF-1).** Identical queries hit the database every time.
- **Development HMAC secret fallback (SC-1).** Should be a hard failure in production.
- **CORS allows any origin (SC-4).** Acceptable for development only.
- **No Prometheus metrics (OP-1).** Metrics are log-based only.
- **Scale uncertainty.** Tested with ∼3,500 triples; behavior at 100K+ untested.

D. What the Audit Validated

The DEP audit validated several architectural decisions:

- **Feature-gated layering** (Architecture: 9/10) — the ability to compile three different profiles was called "exemplary."
- **Type-driven security** (Security: 9/10) — the `AdmissibleEvidenceBundle` pattern was described as "the strongest security implementation I've seen in a sub-15KLOC codebase."
- **Documentation density** (Documentation: 9/10) — noted as exceeding most 100K-line projects.
- **Zero unsafe blocks** (Code Quality: 8/10) — the entire codebase uses safe Rust.

X. DISCUSSION

A. The Provenance Gap in AI Infrastructure

Our evaluation reveals a significant gap: no widely-deployed system provides deterministic, cryptographically-verifiable context windows for autonomous agents. This is not because the need is unrecognized—the AI safety community has extensively discussed auditable AI [42], [43]—but because the infrastructure has not been built.

The Graph Kernel addresses this gap by making provenance a first-class architectural concern. The `AdmissibleEvidenceBundle` type, the HMAC-signed admissibility token, and the policy governance framework together ensure that every context window carries a complete provenance chain.

B. Formal Properties of the Provenance Engine Category

A provenance engine satisfying our four invariants provides three emergent properties:

Regulatory compliance. As AI regulation evolves (EU AI Act [45], NIST AI RMF [46]), the ability to audit which information influenced an agent's decision becomes a compliance requirement.

Multi-agent trust. In systems where multiple agents collaborate, provenance tokens enable trust delegation: Agent A can verify that Agent B's context window was authorized by a shared policy authority without accessing the authority's signing secret.

Deterministic debugging. When an agent produces an unexpected output, deterministic context slicing enables exact reproduction of input conditions—same anchor, same policy, same graph state yields the same context window.

C. Limitations

No semantic search capability. The Graph Kernel achieves 0.42 average relevance on fuzzy/semantic queries. This is a fundamental limitation of exact field matching and motivates the hybrid architecture (§VI).

Scale constraints. The evaluated corpus contains 3,502 triples. For billion-scale applications, purpose-built distributed databases remain necessary.

Relevance metric limitations. Our relevance metric does not capture structural quality. As demonstrated in §VII, keyword search and Graph Kernel can achieve identical 1.00 relevance while producing qualitatively different results (coincidence piles vs. causal chains). Future work should incorporate structural metrics: chain validity, provenance completeness, and causal coherence.

Single-user scope. The current system serves a single user's knowledge graph. Federation requires the protocol extensions described in §XI.

HMAC key management. No key rotation mechanism exists. Key versioning with dual-key verification is needed for production deployments.

RAG++ deployment gap. The RAG++ Rust core contains sophisticated algorithms but the deployed FastAPI service uses vector-only search, explaining the 0.70 relevance. Activating existing algorithms is projected to raise relevance to 0.90+.

XI. Evolution Roadmap

A. Evolution 1: Optimization (3.5 weeks)

SQLite Backend Migration. Implement `SqliteGraphStore` as a native Rust backend using SQLite with WAL mode. Reads from local SQLite (∼5ms), writes propagate to Supabase asynchronously. Projected latency improvement: 291ms → 5–15ms. Feature-gated via `sqlite` Cargo feature.

Query Caching. LRU cache keyed on query parameters with TTL-based invalidation, using existing `lru` + `parking_lot` infrastructure.

Activate RAG++ Hybrid Search. Wire existing `HybridQueryEngine` and `Reranker` through to FastAPI endpoints. Default: RRF fusion (k=60) of HNSW + BM25. Projected RAG++ relevance: 0.70 → 0.90+.

Projected DEP score after Evo 1: 8.6/10 (+1.2).

B. Evolution 2: Expansion (4 weeks)

WebSocket Subscriptions. Real-time push notifications when triples are created, updated, or deleted. Using Axum's built-in WebSocket support with `tokio::sync::broadcast`.

Graph Visualization Endpoint. `GET /api/knowledge/graph` returning D3-compatible JSON, Mermaid syntax, or Graphviz DOT format. Reuses the server-side traversal with output rendering.

Temporal Versioning. `valid_from` and `valid_until` columns on triples, with supersession tracking. Enables temporal queries and knowledge lifecycle management.

Community Detection. Louvain/Leiden algorithm over the knowledge graph topology to identify hierarchical entity clusters.

Projected DEP score after Evo 2: 9.1/10 (+0.5).

C. Evolution 3: Transformation (3–6 months)

Federated Graph Kernel. Multiple Graph Kernel instances (per-user, per-team) share and merge knowledge while preserving provenance boundaries. Federation tokens extend the HMAC model to multi-party verification.

Universal Context Authority. Framework-agnostic context API accepting requests from any agent system (LangChain, CrewAI, AutoGen, custom). Token budget-aware context construction with pluggable output formats.

Graph Kernel SDK. First-party client libraries for Rust, Python (`pip install graph-kernel`), and JavaScript/TypeScript (`npm install @openclaw/graph-kernel`).

W3C PROV-DM Integration. Map the Graph Kernel's provenance model to W3C PROV [40], enabling export to PROV-compliant systems.

Graph Kernel Protocol (GKP/1). Open protocol specification for provenance-governed context authorities with conformance test suite.

Graph Kernel Cloud. Multi-tenant managed service with per-tenant SQLite isolation, JWT authentication, and usage-based billing.

XII. Conclusion

We have presented the Graph Kernel, a deterministic provenance engine that introduces a new category of infrastructure for autonomous AI agent systems. Through evaluation across 27 queries and four retrieval methods, we demonstrate that it achieves perfect relevance on multi-hop structural queries while providing properties no existing system offers: HMAC-signed deterministic context windows, type-level admissibility enforcement, and policy-governed context expansion.

The DEP audit process proved valuable: systematic assessment across 12 engineering dimensions identified 47 specific findings, of which 10 critical fixes were implemented, raising

the projected health score from 7.4 to 8.4/10. The most impactful improvements—native Rust entity normalization, server-side multi-hop traversal, and parameterized SQL queries—addressed real correctness and performance issues that would have affected production reliability.

The hybrid architecture with RAG++ demonstrates that the provenance engine is not a replacement for semantic retrieval but a complementary layer. Where RAG++ finds *what sounds relevant*, the Graph Kernel proves *what is structurally connected and authorized*. Together, they provide the semantic-plus-structural retrieval backbone that consequential agent decisions require.

The Graph Kernel is not a general-purpose search engine (0.84 overall relevance vs. keyword's 0.96), nor a general-purpose graph database (REST filters vs. Cypher/SPARQL), nor a semantic retrieval system (0.42 fuzzy relevance). It is a **provenance engine**—and in that role, among the eleven systems we evaluated, it is irreplaceable.

As autonomous AI agents assume greater responsibility in production systems, the infrastructure to ensure their decisions are traceable, reproducible, and verifiable becomes not optional but essential. The Graph Kernel, and the provenance engine category it establishes, is a step toward that infrastructure.

Appendix A
Reproducibility Statement

All benchmark scripts, raw results, and evaluation code are available in the OpenClaw CompCore repository at `benchmarks/run_benchmark.py`. The Graph Kernel binary can be compiled from source with `cargo build --release --features service`. The evaluation corpus (3,502 triples) is stored in the Supabase PostgreSQL instance; a SQLite snapshot is maintained at `~/.compcore/graph-kernel.db`. Complete benchmark results in JSON format are archived at `/tmp/benchmark_results.json`. The codebase compiles cleanly with all 29 tests passing (14 atlas integration, 13 golden determinism, 2 doc tests).

Appendix B
Corpus Statistics

TABLE XV
Corpus Statistics

Metric	Value
Total triples	3,502
Unique subjects	221
Unique predicates	88
Top predicate	has_file (810, 23.1%)
Data sources	kimi-k2-extraction, topology-ingester
Average confidence	0.73 (Kimi), 0.95 (topology)
RAG++ corpus	107K+ conversation turns
RAG++ bridges	Clawdbot (33K), Claude Code (43K)

TABLE XVI
DEP Audit Scores by Dimension

Dimension	Weight	Score	Weighted
Code Quality	10%	8/10	0.80
Architecture	15%	9/10	1.35
Performance	15%	5/10	0.75
Data Model	10%	7/10	0.70
Security	10%	9/10	0.90
Testing	8%	8/10	0.64
Documentation	5%	9/10	0.45
Dependencies	5%	7/10	0.35
Operational	7%	7/10	0.49
Scale	7%	6/10	0.42
Known Bugs	4%	6/10	0.24
Missing Features	4%	5/10	0.20
Overall	**100%**		**7.4/10**

Appendix C
DEP Audit Score Summary

Post-fix projected score: **8.4/10**.

Appendix D
Post-Audit Changes Summary

TABLE XVII
Post-Audit Implementation Status

Item	Finding	Sev.	Status
Entity normalization	DM-1, BG-1	HIGH	✓
SQL parameterization	AR-1, BG-3	MED	✓
Server-side traversal	MF-1	CRIT	✓
Pool optimization	PF-4, PF-5	MED	✓
Pagination fix	BG-2	MED	✓
DELETE endpoint	MF-7	LOW	✓
Request timeout	OP-3	LOW	✓
Remove async-trait	DP-1	LOW	✓
Object index	DM-2	MED	✓
SQLite backend	PF-1, PF-2	HIGH	Planned
Query caching	PF-1	HIGH	Planned
Prometheus metrics	OP-1	MED	Planned
HMAC key rotation	SC-3	LOW	Planned

References

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 30, 2017.

[2] B. Mohr, E. Bueno de Mesquita, and S. van Cranenburgh, "Weaviate: An open-source vector database," 2023. [Online]. Available: https://weaviate.io

[3] J. J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with GPUs," *IEEE Transactions on Big Data*, vol. 7, no. 3, pp. 535–547, 2021.

[4] P. Lewis *et al.*, "Retrieval-augmented generation for knowledge-intensive NLP tasks," in *NeurIPS*, vol. 33, 2020, pp. 9459–9474.

[5] Y. Gao *et al.*, "Retrieval-augmented generation for large language models: A survey," *arXiv:2312.10997*, 2023.

[6] Neo4j, Inc., "Neo4j Graph Database," 2024. [Online]. Available: https://neo4j.com

[7] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. Reutter, and D. Vrgoč, "Foundations of modern query languages for graph databases," *ACM Computing Surveys*, vol. 50, no. 5, pp. 1–40, 2017.

[8] Zep, "Zep: Long-term memory for AI assistants," 2024. [Online]. Available: https://www.getzep.com

[9] J. F. Sowa, "Semantic networks," in *Encyclopedia of Artificial Intelligence*, 2nd ed., S. C. Shapiro, Ed. Wiley, 1992.

[10] A. Singhal, "Introducing the Knowledge Graph: Things, not strings," Google Official Blog, May 2012.

[11] Apache Software Foundation, "Apache Jena," 2024. [Online]. Available: https://jena.apache.org

[12] Blazegraph, "Blazegraph Database," 2024. [Online]. Available: https://blazegraph.com

[13] N. Francis *et al.*, "Cypher: An evolving query language for property graphs," in *SIGMOD*, 2018, pp. 1433–1445.

[14] S. Ji, S. Pan, E. Cambria, P. Marttinen, and P. S. Yu, "A survey on knowledge graphs: Representation, acquisition, and applications," *IEEE Trans. NNLS*, vol. 33, no. 2, pp. 494–514, 2022.

[15] A. Hogan *et al.*, "Knowledge graphs," *ACM Computing Surveys*, vol. 54, no. 4, pp. 1–37, 2021.

[16] A. Hogan *et al.*, "Knowledge graphs," *Synthesis Lectures on Data, Semantics, and Knowledge*, Morgan & Claypool, 2022.

[17] K. Guu, K. Lee, Z. Tung, P. Pasupat, and M.-W. Chang, "Retrieval augmented language model pre-training," in *ICML*, 2020, pp. 3929–3938.

[18] G. Izacard and E. Grave, "Leveraging passage retrieval with generative models for open domain question answering," in *EACL*, 2021, pp. 874–880.

[19] A. Asai, Z. Wu, Y. Wang, A. Sil, and H. Hajishirzi, "Self-RAG: Learning to retrieve, generate, and critique through self-reflection," in *ICLR*, 2024.

[20] S.-C. Yan, S.-W. Gu, L.-Y. Wu, and S. Singh, "Corrective retrieval augmented generation," *arXiv:2401.15884*, 2024.

[21] W. X. Zhao, J. Liu, R. Ren, and J.-R. Wen, "Dense text retrieval based on pretrained language models: A survey," *ACM Computing Surveys*, vol. 56, no. 9, 2024.

[22] Pinecone Systems, "Pinecone: Vector Database for Machine Learning," 2024. [Online]. Available: https://www.pinecone.io

[23] Chroma, "Chroma: The AI-native open-source embedding database," 2024. [Online]. Available: https://www.trychroma.com

[24] Qdrant, "Qdrant: Vector similarity search engine," 2024. [Online]. Available: https://qdrant.tech

[25] Milvus, "Milvus: Open-source vector database for AI," 2024. [Online]. Available: https://milvus.io

[26] D. Edge *et al.*, "From local to global: A graph RAG approach to query-focused summarization," *arXiv:2404.16130*, 2024.

[27] V. A. Traag, L. Waltman, and N. J. van Eck, "From Louvain to Leiden: Guaranteeing well-connected communities," *Scientific Reports*, vol. 9, p. 5233, 2019.

[28] Z. Guo, L. Shang, J. Jiang, and J. X. Huang, "LightRAG: Simple and fast retrieval-augmented generation," *arXiv:2410.05779*, 2024.

[29] A. Soman, P. G. Bhat, V. Barber, and S. Iyengar, "Biomedical knowledge graph-enhanced prompt generation for large language models," *arXiv:2311.17330*, 2023.

[30] B. Gutiérrez, Y. Yang, H. Gu, M. Sami, and Y. Su, "HippoRAG: Neurobiologically inspired long-term memory for large language models," in *NeurIPS*, 2024.

[31] J. Liu, "LlamaIndex: A data framework for LLM applications," 2024. [Online]. Available: https://www.llamaindex.ai

[32] H. Chase, "LangChain," 2024. [Online]. Available: https://www.langchain.com

[33] I. Beltagy, M. E. Peters, and A. Cohan, "Longformer: The long-document transformer," *arXiv:2004.05150*, 2020.

[34] M. Zaheer *et al.*, "Big Bird: Transformers for longer sequences," in *NeurIPS*, vol. 33, 2020, pp. 17283–17297.

[35] W. Zhong, L. Guo, Q. Gao, H. Ye, and Y. Wang, "MemoryBank: Enhancing large language models with long-term memory," in *AAAI*, vol. 38, 2024.

[36] A. Bulatov, Y. Kuratov, and M. S. Burtsev, "Recurrent memory transformer," in *NeurIPS*, vol. 35, 2022, pp. 11079–11091.

[37] C. Packer, S. Wooders, K. Lin, V. Fang, S. Patil, I. Stoica, and J. E. Gonzalez, "MemGPT: Towards LLMs as operating systems," *arXiv:2310.08560*, 2023.

[38] P. Buneman, S. Khanna, and W.-C. Tan, "Why and where: A characterization of data provenance," in *ICDT*, 2001, pp. 316–330.

[39] J. Cheney, L. Chiticariu, and W.-C. Tan, "Provenance in databases: Why, how, and where," *Found. Trends Databases*, vol. 1, no. 4, pp. 379–474, 2009.

[40] L. Moreau, P. Missier, *et al.*, "PROV-DM: The PROV Data Model," W3C Recommendation, 2013. [Online]. Available: https://www.w3.org/TR/prov-dm/

[41] R. Liang, C. Niu, F. Zhang, Z. Qi, and Y. Hao, "Blockchain-based data provenance for AI: A comprehensive survey," *IEEE Access*, vol. 11, pp. 61748–61770, 2023.

[42] D. Amodei *et al.*, "Concrete problems in AI safety," *arXiv:1606.06565*, 2016.

[43] S. Weidinger *et al.*, "Ethical and social risks of harm from language models," *arXiv:2112.04359*, 2021.

[44] M. Mitchell *et al.*, "Model cards for model reporting," in *FAccT*, 2019, pp. 220–229.

[45] European Parliament, "Regulation (EU) 2024/1689 laying down harmonised rules on artificial intelligence (AI Act)," *Official Journal of the European Union*, 2024.

[46] National Institute of Standards and Technology, "AI Risk Management Framework (AI RMF 1.0)," NIST AI 100-1, January 2023.

[47] M. Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT)," RFC 7519, IETF, 2015.

[48] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica, "Friday: Global comprehension for distributed replay," in *NSDI*, 2007, pp. 285–298.

[49] O. Laadan, N. Viennot, and J. Nieh, "Transparent, lightweight application execution replay on commodity multiprocessor operating systems," in *SIGMETRICS*, 2010, pp. 155–166.

[50] D. Tolnay *et al.*, "Axum: Ergonomic and modular web framework built with Tokio," 2024. [Online]. Available: https://github.com/tokio-rs/axum

[51] C. Lerche *et al.*, "Tokio: An asynchronous runtime for Rust," 2024. [Online]. Available: https://tokio.rs

[52] Amazon Web Services, "Amazon Neptune," 2024. [Online]. Available: https://aws.amazon.com/neptune/

[53] Dgraph Labs, "Dgraph: Distributed graph database," 2024. [Online]. Available: https://dgraph.io

[54] Vaticle, "TypeDB: The polymorphic database," 2024. [Online]. Available: https://typedb.com

[55] F. Petroni *et al.*, "Language models as knowledge bases?" in *EMNLP-IJCNLP*, 2019, pp. 2463–2473.

[56] Z. Jiang, F. F. Xu, J. Araki, and G. Neubig, "How can we know what language models know?" *TACL*, vol. 8, pp. 423–438, 2020.

[57] J. Wei *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," in *NeurIPS*, vol. 35, 2022, pp. 24824–24837.

[58] T. Schick *et al.*, "Toolformer: Language models can teach themselves to use tools," in *NeurIPS*, vol. 36, 2024.

[59] S. Yao *et al.*, "ReAct: Synergizing reasoning and acting in language models," in *ICLR*, 2023.

[60] N. Shinn, F. Cassano, A. Gopinath, K. R. Narasimhan, and S. Yao, "Reflexion: Language agents with verbal reinforcement learning," in *NeurIPS*, vol. 36, 2024.