SHSHELL.COM

RESEARCH REPORT

BY SUDEEP DEVKOTA

MAY 2026

https://shshell.com

SHSHELL.COM

KNOWLEDGE GRAPHS AND GRAPH RAG: A PRACTICAL IMPLEMENTATION GUIDE

A builder-focused report on knowledge graphs, Graph RAG architecture, retrieval patterns, evaluation, and the step-by-step path to shipping Graph RAG in production applications.

RESEARCH REPORT

BY SUDEEP DEVKOTA

MAY 2026

https://shshell.com

Contents

Knowledge Graphs and Graph RAG: A Practical Implementation Guide

> **EXECUTIVE INSIGHT**
>
> *Graph RAG is not "vector search with a graph database." It is a retrieval architecture that adds explicit entities, relationships, graph traversal, and dataset-level summaries to the normal RAG loop. Use it when the answer depends on connections, lineage, aggregation, or multi-hop reasoning.*

Executive Summary

Retrieval-augmented generation gave teams a practical way to connect large language models to private documents. The first wave of RAG systems mostly followed the same pattern: split documents into chunks, embed the chunks, retrieve the nearest vectors, and ask the model to answer using those snippets. That pattern still works for many support bots, policy assistants, and document Q&A products.

Graph RAG becomes valuable when the question is not answered by one nearby paragraph. It helps when users ask about relationships across people, products, documents, incidents, contracts, code modules, customers, claims, or events. A knowledge graph gives the application a structured memory: entities become nodes, relationships become edges, source chunks remain attached as evidence, and retrieval can combine semantic search with graph traversal.

This report explains what knowledge graphs are, why Graph RAG matters, and how to implement it in an application without turning the project into a research lab. The practical architecture is hybrid: keep vector search for fuzzy discovery, add a graph for relationships and explainability, use community summaries for global questions, and evaluate the system with tests that measure groundedness, completeness, and path quality.

Why Naive RAG Breaks

Naive RAG retrieves text by similarity. That is useful when the user's question resembles the wording of the source material. It struggles when the question requires:

- **Multi-hop reasoning**: "Which suppliers are connected to the delayed part through the impacted product line?"
- **Disambiguation**: "Is Apple the customer, the vendor, or the product family in this contract?"
- **Dataset-wide themes**: "What are the top recurring failure modes across all support escalations this quarter?"
- **Lineage and provenance**: "Which policy changed because of which incident, and who approved it?"
- **Precise relationships**: "Which services depend on this database and are owned by the platform team?"

Vector search sees semantic closeness. A graph sees structure. Production Graph RAG uses both.

What Is a Knowledge Graph?

A knowledge graph stores facts as connected entities. A minimal graph is made of:

- **Nodes**: entities such as Person, Company, Product, Feature, Contract, Ticket, Regulation, Repository, Service, or Incident.
- **Edges**: typed relationships such as OWNS, DEPENDS*ON, APPROVED*BY, VIOLATES, MENTIONS, SHIPPED*IN, BLOCKS, TREATS, or CAUSED*BY.
- **Properties**: attributes on nodes or edges, such as date, confidence, source_id, owner, status, amount, jurisdiction, or version.
- **Evidence links**: references back to the chunks, files, tables, tickets, emails, or records where the fact came from.

For Graph RAG, the graph should not be treated as a decorative visualization. It is a retrieval index. The graph exists so the application can answer questions by following relationships, ranking paths, resolving entities, and citing evidence.

What Is Graph RAG?

Graph RAG is retrieval-augmented generation using a graph-shaped index. The common production pattern has four layers:

1. **Document layer**: raw text, tables, PDFs, tickets, web pages, transcripts, or database records.
2. **Vector layer**: embeddings for chunks, entities, and sometimes community summaries.
3. **Graph layer**: extracted entities, typed relationships, properties, source links, and graph communities.
4. **Generation layer**: the LLM receives retrieved chunks, graph facts, paths, and citations, then produces a grounded answer.

Microsoft's GraphRAG work popularized the distinction between local and global questions. Local search uses entities, relationships, and source chunks around a specific topic. Global search uses community reports to answer questions about the entire dataset. Neo4j's GraphRAG tooling and LangChain integrations make the same idea practical for application builders: combine vector retrieval, graph traversal, and prompt-grounded generation.

Retrieval Patterns That Matter

Pattern 1: Vector First, Graph Expand

Use vector search to find the most relevant chunks or entities, then expand from those nodes through the graph.

Best for:

- Product documentation assistants
- Engineering runbooks
- Customer support knowledge bases
- Legal and compliance Q&A

Implementation idea:

```
query -> vector search -> seed chunks/entities -> graph traversal -> evidence pack -
> LLM answer
```

The vector stage finds the neighborhood. The graph stage adds the surrounding facts that the nearest chunks would miss.

Pattern 2: Entity First, Path Search

Extract named entities from the user question, resolve them to canonical graph nodes, then search for paths between them.

Best for:

- Dependency questions
- Risk and impact analysis
- Relationship-heavy domains
- "How is X connected to Y?" questions

Implementation idea:

```
query -> entity extraction -> entity resolution -> shortest paths / typed paths -> evidence pack -> LLM answer
```

This pattern is especially useful when the answer is a chain of relationships, not a paragraph.

Pattern 3: Text-to-Cypher or Text-to-Graph-Query

For structured questions, the model can translate a natural-language question into a graph query. This is powerful but should be constrained.

Best for:

- Known schemas
- Internal analytics
- Expert users
- Applications where the graph schema is stable

Guardrails:

- Give the model a schema, not the whole database.
- Use read-only credentials.
- Validate generated queries before execution.
- Limit query complexity and result size.
- Return source evidence with query results.

Pattern 4: Community Summaries for Global Search

For questions that ask about themes, trends, or whole-dataset meaning, community summaries can outperform raw top-k vector search. The graph is clustered into communities. Each community gets an LLM-generated report. The query engine retrieves or ranks relevant reports, then uses a map-reduce style synthesis step.

Best for:

- "What are the major themes?"
- "Summarize the last month of customer complaints."
- "What risks show up across all vendor contracts?"
- "What patterns are emerging in incident reports?"

Microsoft's GraphRAG documentation describes global search as a way to reason over community reports rather than isolated chunks. Newer approaches such as DRIFT search combine global and local retrieval to improve quality and efficiency.

Reference Architecture

```
flowchart TD
    A[Source Systems] --> B[Ingestion Pipeline]
    B --> C[Chunking and Metadata]
    C --> D[Embeddings]
    C --> E[Entity and Relationship Extraction]
    D --> F[Vector Index]
    E --> G[Entity Resolution]
    G --> H[Knowledge Graph]
    H --> I[Community Detection]
    I --> J[Community Reports]
    K[User Query] --> L[Query Router]
    L --> M[Vector Retrieval]
    L --> N[Graph Traversal]
    L --> O[Global Community Search]
    M --> P[Evidence Pack]
    N --> P
    O --> P
    P --> Q[Grounded LLM Answer]
    Q --> R[Citations, Paths, and Follow-up Questions]
```

Data Model Blueprint

Start with a small schema. Most failed Graph RAG projects fail because the first ontology is too ambitious.

Node Type	Purpose	Example Properties
Document	Source file or record	id, title, source*system, created*at, owner

Node Type	Purpose	Example Properties
Chunk	Text segment used for citations	text, page, section, embedding_id
Entity	Canonical business object	name, type, aliases, confidence
Topic	Theme or cluster label	name, summary, confidence
UserQuery	Optional observability object	query*text*, *user*id_hash, timestamp

Edge Type	Meaning
CONTAINS	Document contains Chunk
MENTIONS	Chunk mentions Entity
RELATED_TO	Entity has a general relationship to Entity
DEPENDS_ON	Service, process, or component dependency
OWNED_BY	Ownership or responsibility
EVIDENCED_BY	Entity or relationship links back to Chunk
BELONGS*TO*COMMUNITY	Entity belongs to a graph community

The first version should prefer fewer node types, clear relationship names, and high-quality evidence links.

Implementation Path

Step 1: Pick the Use Case

Choose one workflow where relationships matter. Good first projects include:

- Support escalation intelligence
- Contract obligation discovery
- Engineering dependency Q&A

- Sales account research
- Policy and compliance assistants
- Clinical or scientific literature exploration

Avoid starting with "graph all company knowledge." That is a program, not a sprint.

Step 2: Build the Ingestion Pipeline

Ingestion should preserve source structure. Keep document IDs, page numbers, headings, timestamps, access-control labels, and source-system URLs.

Minimum pipeline:

```
load -> normalize -> chunk -> enrich metadata -> embed chunks -> extract entities/relations -> write graph -> write vector index
```

For unstructured documents, use extraction prompts with a schema. For structured data, use deterministic transforms first and LLM extraction only where the source is ambiguous.

Step 3: Extract Entities and Relationships

LLMs can extract triples in the form:

```
subject -> relationship -> object
```

Example:

```
"The Billing API depends on Customer Profile Service for account validation."

Entity: Billing API, type Service
Entity: Customer Profile Service, type Service
Relationship: Billing API DEPENDS_ON Customer Profile Service
Evidence: source chunk id
Confidence: 0.86
```

Extraction should return structured JSON, not prose.

```
{
  "entities": [
    {"name": "Billing API", "type": "Service"},
    {"name": "Customer Profile Service", "type": "Service"}
  ],
  "relationships": [
    {
      "source": "Billing API",
```

```
      "type": "DEPENDS_ON",
      "target": "Customer Profile Service",
      "evidence": "chunk_0018",
      "confidence": 0.86
    }
  ]
}
```

Step 4: Resolve Entities

Entity resolution decides whether "OpenAI", "Open AI", and "OpenAI Inc." are the same node. This step is critical.

Use a layered strategy:

- Exact match on stable identifiers when available.
- Alias table for known names.
- Embedding similarity for candidate generation.
- LLM or rules-based adjudication for ambiguous matches.
- Human review for high-impact domains.

Store aliases and confidence scores. Never silently merge low-confidence entities.

Step 5: Store the Graph and Vectors

Common options:

- **Neo4j** for property graphs, Cypher queries, graph traversal, and GraphRAG ecosystem support.
- **Postgres plus pgvector** for teams that want one operational database and a lighter graph model.
- **NetworkX** for prototypes and offline analytics.
- **Managed vector databases** such as Pinecone, Qdrant, Weaviate, or cloud-native vector search for the vector layer.

For many teams, Neo4j plus a vector index is the most straightforward Graph RAG starting point because the graph, vector retrieval, and Cypher-based traversal patterns can live close together.

Step 6: Build the Query Router

Route the question based on intent.

Query Type	Example	Retrieval Strategy
Factual	"What does the refund policy say?"	Vector search
Entity-specific	"What do we know about Vendor A?"	Entity lookup plus local graph expansion
Relationship	"How is Service A connected to Database B?"	Entity resolution plus path search
Impact	"What breaks if this API changes?"	Traversal across DEPENDS_ON edges
Global	"What themes appear across complaints?"	Community reports plus synthesis

The router can be a small classifier prompt or deterministic rules with an LLM fallback.

Step 7: Create the Evidence Pack

The evidence pack is the object given to the LLM. It should include:

- User question
- Retrieved chunks
- Graph facts
- Relationship paths
- Source citations
- Confidence scores
- Retrieval method used
- Instructions to answer only from evidence

Example:

```
Question:
What services are affected if Customer Profile Service is unavailable?

Graph paths:
Billing API -DEPENDS_ON-> Customer Profile Service
Checkout Service -CALLS-> Billing API -DEPENDS_ON-> Customer Profile Service
Fraud Review -READS_FROM-> Customer Profile Service

Source chunks:
[runbook.md#L42] Billing API validates accounts through Customer Profile Service.
```

```
[architecture.md#L88] Checkout invokes Billing API during purchase authorization.
```

Step 8: Generate the Answer

Use a strict generation prompt:

```
Answer the user's question using only the evidence below.
When the answer depends on a graph path, show the path.
When evidence is weak or missing, say what is missing.
Cite source chunks by id.
Do not invent entities, relationships, dates, owners, or metrics.
```

Step 9: Evaluate Before Launch

Evaluate more than final answer quality. Graph RAG has more moving parts than naive RAG.

Layer	What to Test
Extraction	Precision and recall of entities and relationships
Resolution	Duplicate rate, false merges, alias handling
Retrieval	Whether the correct nodes, paths, and chunks are retrieved
Generation	Groundedness, citation accuracy, answer completeness
Security	Access-control filtering before retrieval and generation
Operations	Ingestion cost, latency, graph growth, stale facts

Use a benchmark set with expected paths, not just expected paragraphs.

Minimal Python Sketch

The following sketch shows the shape of a Graph RAG service. It is intentionally vendor-neutral pseudocode.

```
def answer(question: str, user_context: dict) -> dict:
    intent = classify_query(question)

    if intent == "global":
        reports = retrieve_community_reports(question, top_k=12)
        evidence = build_evidence_pack(question, community_reports=reports)

    elif intent == "relationship":
        entities = extract_query_entities(question)
        resolved = resolve_entities(entities)
        paths = graph.find_relevant_paths(resolved, max_hops=3)
        chunks = fetch_source_chunks(paths)
        evidence = build_evidence_pack(question, paths=paths, chunks=chunks)

    else:
        seeds = vector.search(question, top_k=8, filters=user_context["acl"])
        expanded = graph.expand_from_chunks(seeds, max_hops=2)
        chunks = rerank_chunks(seeds + expanded.source_chunks, question)
        evidence = build_evidence_pack(question, chunks=chunks, graph_facts=expanded
.facts)

    answer_text = llm.generate(GROUNDED_ANSWER_PROMPT, evidence)
    return {
        "answer": answer_text,
        "citations": evidence.citations,
        "retrieval_mode": intent,
    }
```

Neo4j-Oriented Implementation Notes

Neo4j's official GraphRAG package for Python provides first-party features for knowledge graph construction, retrievers, and RAG pipelines. The package supports modern Python versions and can integrate with LLM providers such as OpenAI, Anthropic, Google, Cohere, Mistral, and Ollama through optional extras.

A typical Neo4j implementation uses:

- A graph database for entities and relationships.
- A vector index for chunks or entity descriptions.
- A graph-enhanced retriever that starts from vector results and expands across relationships.
- A prompt template that includes both text chunks and graph facts.

Example Cypher-shaped context:

```
MATCH (chunk:Chunk)<-[:EVIDENCED_BY]-(entity:Entity)-[r*1..2]-(neighbor:Entity)
WHERE chunk.id IN $seed_chunk_ids
RETURN chunk.text AS source_text,
       entity.name AS entity,
       neighbor.name AS neighbor,
       r AS relationship_path
LIMIT 50
```

The exact schema will vary, but the principle should not: retrieve text and structure together.

Security and Governance

Graph RAG can accidentally leak more than naive RAG because relationships cross document boundaries. Treat access control as a retrieval-time requirement, not a UI filter.

Production rules:

- Apply ACL filters before vector search and before graph traversal.
- Do not traverse from an allowed node into a forbidden node.
- Store source permissions on documents, chunks, and sensitive entities.
- Redact secrets and regulated data during ingestion.
- Log retrieved evidence, not just generated answers.
- Keep graph updates auditable.
- Separate extraction confidence from truth.

In regulated domains, relationships may be as sensitive as documents. "Person A investigated Company B" can be more sensitive than either node alone.

Common Failure Modes

Over-extraction

The LLM creates too many entities and relationships. The graph becomes noisy.

Fix:

- Use a narrow schema.

- Require evidence spans.
- Set confidence thresholds.
- Review high-impact relation types.

False Entity Merges

Two different entities become one node.

Fix:

- Use stable IDs where possible.
- Track aliases separately.
- Require human review for risky merges.

Pretty Graph, Weak Retrieval

The graph looks impressive but does not improve answers.

Fix:

- Test retrieval paths against real questions.
- Add query router logic.
- Tune traversal depth and edge types.
- Measure answer lift over vector-only RAG.

Global Search Cost Explosion

Community report synthesis becomes expensive.

Fix:

- Cache community reports.
- Use smaller models for ranking and relevance classification.
- Retrieve relevant communities before map-reduce.
- Refresh summaries incrementally.

Production Checklist

- Define one high-value use case.
- Create a small graph schema.

- Preserve source IDs and permissions.
- Build chunk embeddings.
- Extract entities and relationships as JSON.
- Resolve entities with confidence.
- Store evidence links for every relationship.
- Implement vector-first graph expansion.
- Implement entity-first path search.
- Add community summaries only when global questions matter.
- Build a query router.
- Ground answers in evidence packs.
- Evaluate extraction, retrieval, paths, citations, and final answers.
- Monitor cost, latency, graph growth, stale facts, and feedback.

Build vs. Buy Decision

Choice	Best When	Tradeoff
Naive RAG only	Simple document Q&A	Weak relationship reasoning
Managed Graph RAG package	Need faster implementation	Less control over internals
Neo4j GraphRAG	Need graph traversal, Cypher, and mature graph tooling	Requires graph modeling discipline
Custom graph plus vector stack	Need deep product-specific control	More engineering and evaluation work
Microsoft GraphRAG style indexing	Need whole-corpus global summarization	Higher indexing and summarization cost

Recommended First 30 Days

Week 1: Scope

- Pick one workflow.
- Collect 50 representative questions.
- Label which questions are vector, entity, relationship, impact, or global.
- Draft the first schema.

Week 2: Ingestion

- Load 100 to 1,000 documents or records.
- Chunk and embed.
- Extract entities and relationships.
- Store evidence links.

Week 3: Retrieval

- Build vector-only baseline.
- Add vector-first graph expansion.
- Add entity-first path search.
- Compare answers against expected evidence.

Week 4: Productization

- Add access control.
- Add citations and path display.
- Add feedback collection.
- Add evaluation tests to CI.
- Decide whether community summaries are needed.

SEO Summary

Knowledge graphs and Graph RAG help AI applications answer questions that require entity relationships, source-grounded reasoning, and whole-dataset understanding. To implement Graph RAG, combine document chunking, embeddings, entity extraction, relationship extraction, entity resolution, a graph database, vector retrieval, graph traversal, community summaries, and grounded LLM generation. The most practical production architecture is hybrid vector plus graph retrieval with strong evaluation and access-control enforcement.

Sources and Further Reading

- Microsoft Research, "GraphRAG: New tool for complex data discovery now on GitHub": https://www.microsoft.com/en-us/research/blog/graphrag-new-tool-for-complex-data-discovery-now-on-github/
- Microsoft Research, "Introducing DRIFT Search: Combining global and local search methods to improve quality and efficiency": https://www.microsoft.com/en-us/research/blog/introducing-drift-search-combining-global-and-local-search-methods-to-improve-quality-and-efficiency/
- Microsoft Research, "GraphRAG: Improving global search via dynamic community selection": https://www.microsoft.com/en-us/research/blog/graphrag-improving-global-search-via-dynamic-community-selection/
- Microsoft GraphRAG documentation, "Query Engine": https://microsoft.github.io/graphrag/query/overview/
- Microsoft GraphRAG documentation, "Global Search": https://microsoft.github.io/graphrag/query/global_search/
- Microsoft GraphRAG documentation, "Local Search": https://microsoft.github.io/graphrag/query/local_search/
- Neo4j GraphRAG for Python documentation: https://neo4j.com/docs/neo4j-graphrag-python/current/
- Neo4j developer guide, "Neo4j GraphRAG Python Package": https://neo4j.com/developer/genai-ecosystem/graphrag-python/
- Neo4j PyPI package, "neo4j-graphrag": https://pypi.org/project/neo4j-graphrag/

ShShell.com
https://shshell.com