**GenAI + RDF Triple Store + Neo4j + Vector DB — Text Diagrams & Details**

Generated: 2025-09-01 22:08 UTC

This document consolidates three text-only diagrams—fully editable—and the key details for building a ChatGPT‑style system that combines an RDF triple store (SPARQL), a Neo4j app graph (Cypher + GDS), and a Vector Database (embeddings).

**Diagram 0 — One‑Text Model (High‑Level)**

┌────────────────────────────────────────────────────────────────┐  
 │ DATA (INGEST) │  
 │ Files • DB tables • APIs • Web pages • Partners’ RDF/SPARQL │  
 └──────────────┬───────────────────────────┬─────────────────────┘  
 │ │  
 ┌──────────────▼──────────────┐ ┌────────▼─────────┐  
 │ Normalize & Chunk Text │ │ Extract Entities │  
 │ (clean, split, metadata) │ │ (NER, rules, KB) │  
 └──────────────┬──────────────┘ └────────┬─────────┘  
 │ │  
 ┌────────────────────▼──────────────────┐ ┌────▼──────────────────┐  
 │ RDF Modeling (RDFS/OWL, IRIs, SKOS) │ │ LPG Modeling (Neo4j) │  
 │ + SHACL validation (contracts) │ │ labels/props/edges │  
 └──────────────┬────────────────────────┘ └──────────┬───────────┘  
 │ │  
 ┌──────────────────▼──────────────┐ ┌───────────────▼──────────────┐  
 │ RDF Triple Store (SPARQL) │ │ Neo4j App Graph (Cypher + GDS)│  
 │ • publish /sparql, JSON-LD, DCAT│ │ • fast traversals, algorithms │  
 └─────────────────┬───────────────┘ └───────────────┬──────────────┘  
 │ │  
 │ ┌───────────────────────▼─────────────────────┐  
 │ │ Vector DB / Vector Index (chunks + embeddings)│  
 │ │ • kNN search for semantic similarity │  
 │ └───────────────────────┬─────────────────────┘  
 │ │  
 └──────────────────────────┬───────────────┘  
 │  
 ┌──────────────────▼───────────────────┐  
 │ Query Orchestrator (Router + Tools) │  
 │ • Entity linker (uses SKOS/IRIs) │  
 │ • Route per question: │  
 │ - SPARQL (exact facts/inference) │  
 │ - Cypher (paths/algos, low-latency)│  
 │ - Vector (open-ended/snippets) │  
 └──────────────────┬───────────────────┘  
 │  
 ┌───────────────▼────────────────────┐  
 │ LLM Answer Composer (prompting) │  
 │ • Merges results + citations │  
 │ • Factual guardrails (re-ask/store)│  
 └───────────────┬────────────────────┘  
 │  
 ┌───────────────▼────────────────────┐  
 │ Final Answer │  
 │ • Facts (KG) + Evidence (snippets) │  
 │ • Links/IRIs + provenance │  
 └─────────────────────────────────────┘

**Roles (summary)**

• RDF store = publication & interoperability layer (IRIs, SPARQL, SHACL, JSON‑LD/DCAT).

• Neo4j = serving graph for low‑latency traversals & GDS algorithms; also convenient for graph‑aware RAG.

• Vector DB = semantic text recall (kNN); returns chunks with scores and metadata.

• Orchestrator = detects intent, runs SPARQL/Cypher/Vector paths, merges results, calls LLM for final composition.

**Diagram 1 — Fill the Data Structure (Build‑Time)**

[Sources]  
 PDFs | HTML | DB tables | CSV/JSON | Partner RDF/SPARQL  
 │  
 ▼  
[Normalize]  
 - extract text (Tika/PDFBox)  
 - clean (remove boilerplate)  
 - keep metadata: doc\_id, title, url, date  
 │  
 ▼  
[Chunk Text]  
 - split into ~1–2k chars with overlap  
 - assign: chunk\_id, section\_id, seq\_no  
 │  
 ├───────────────────────────────┐  
 │ │  
 ▼ ▼  
[Entity & Term Extraction] [Embed Chunks]  
 - detect entities (NER/rules) - embedding := model(text)  
 - map to canonical IDs: - store 1536‑D (or model‑dim)  
 • RDF IRIs (SKOS labels) - keep vector with chunk\_id  
 • Neo4j nodes (codes/keys) - add cosine index  
 │ │  
 ├───────────────┬───────────────┘  
 │ │  
 ▼ ▼  
[RDF Modeling] [Neo4j Modeling]  
 - Classes: Document, Section, - Labels: :Document, :Section,  
 Chunk, Entity (+ domain) :Chunk, :Entity (+ domain)  
 - Properties: hasSection, - Rels: :HAS\_SECTION,  
 hasChunk, next, mentions, :HAS\_CHUNK, :NEXT, :MENTIONS,  
 RELATED\_TO, … domain rels (e.g., :BELONGS\_TO)  
 - Shapes: SHACL (contracts) - Constraints: uniqueness on keys  
 │ │  
 ▼ ▼  
[RDF Triple Store] [Neo4j App Graph]  
 - load triples - MERGE nodes/rels  
 - expose /sparql - create vector index on Chunk.embedding  
 │ │  
 └───────┬───────┘  
 ▼  
[Vector DB / Vector Index]  
 - (chunk\_id, embedding, metadata: doc\_id, section\_id, entity\_links)  
 - kNN ready for semantic retrieval  
  
Minimal data structures (examples)  
----------------------------------  
Chunk record:  
 {chunk\_id, doc\_id, section\_id, seq\_no, text,  
 embedding[], source\_url, entity\_ids[]}  
  
Neo4j nodes/rels:  
 (:Document{docId,title,url})-[:HAS\_SECTION]->(:Section{sectionId})  
 (:Section)-[:HAS\_CHUNK]->(:Chunk{chunkId,seq\_no,text})  
 (:Chunk)-[:NEXT]->(:Chunk) ; (:Chunk)-[:MENTIONS]->(:Entity{uri})  
  
RDF triples (Turtle):  
 ex:doc1 ex:hasSection ex:s1 . ex:s1 ex:hasChunk ex:c1 .  
 ex:c1 ex:mentions ex:F\_ACC ; ex:next ex:c2 .

**Build‑Time Notes**

• Keep IRIs stable and versioned; validate RDF with SHACL before loading.

• In Neo4j, create uniqueness constraints on identifiers and a vector index on chunk embeddings.

• Store back‑references (chunk → entities; entity → source docs) for provenance and citations.

**Diagram 2 — Ask → Retrieve → Answer (Run‑Time)**

User Question  
 e.g., "Which 2023 Falcon X trims have ACC and how many sold in CA?"  
 │  
 ▼  
[Query Orchestrator (Router + Tools)]  
 - Normalize question (lang, stopwords)  
 - Entity linking via SKOS/IRIs + Neo4j keys (ACC → ex:F\_ACC; Falcon X → ex:M1)  
 - Intent detection:  
 • Fact/filters/enumeration? → SPARQL path  
 • Path/impact/graph ranking? → Cypher (+ optional GDS)  
 • Open-ended/why/how? → Vector path  
 - May run 2–3 paths and merge  
 │ │ │  
 │ │ └───────────────► [Vector DB / Index]  
 │ │ kNN on chunk embeddings  
 │ │ RETURN top-k {text, score, doc\_id}  
 │ │  
 │ └────────────────────────────► [Neo4j App Graph]  
 │ Cypher traversals/aggregations  
 │ (optionally GDS for ranking/paths)  
 │  
 └─────────────────────────────────────► [RDF Triple Store]  
 SPARQL for exact facts,  
 reasoning, federation  
 │  
 ▼  
[Result Aggregation]  
 - Join facts (SPARQL/Cypher rows) with evidence (vector snippets)  
 - De-duplicate; compute confidence; build provenance (IRIs, node IDs, URLs)  
 │  
 ▼  
[LLM Answer Composer]  
 - Prompt: "Using Facts: {tables} and Evidence: {snippets}, answer concisely,  
 include citations (IRIs, doc URLs), or say 'not found'."  
 │  
 ▼  
[Final Answer]  
 - Structured facts + quoted snippets  
 - Citations: SPARQL IRIs, Neo4j nodes/paths, document URLs

**Tiny Examples (per path)**

SPARQL (RDF facts):

PREFIX ex:<http://example.com/auto#> PREFIX xsd:<http://www.w3.org/2001/XMLSchema#>  
SELECT ?trim (SUM(?q) AS ?units) WHERE {  
 ?trim a ex:Trim ; ex:belongsTo ex:M1 ;  
 ex:modelYear "2023"^^xsd:gYear ; ex:hasFeature ex:F\_ACC .  
 ?sr a ex:SaleRecord ; ex:forTrim ?trim ; ex:soldIn ex:CA ;  
 ex:saleDate "2023-03-15"^^xsd:date ; ex:quantity ?q .  
} GROUP BY ?trim

Cypher (Neo4j paths/aggregation):

MATCH (:VehicleModel {code:'FALCON\_X'})<-[:BELONGS\_TO]-(t:Trim {modelYear:2023})  
MATCH (t)-[:HAS\_FEATURE]->(:Feature {code:'ACC'})  
MATCH (sr:SaleRecord)-[:FOR\_TRIM]->(t)  
MATCH (sr)-[:SOLD\_IN]->(:Region {code:'CA'})  
WHERE date(sr.saleDate)=date('2023-03-15')  
RETURN t.code AS trim, sum(sr.quantity) AS units;

Vector (open‑ended explanation):

query = "Explain ACC benefits for Falcon X drivers in California"  
top\_k = vector\_index.query(k=5, embedding=embed(query))  
# -> {chunk\_id, text, score, doc\_id, entity\_links:[F\_ACC,M1,CA]}