Daniel
JACOBSEN

Production-Grade RAG

A Practical Guide to Building Reliable AI Systems

Daniel
JACOBSEN

Production-Grade RAG

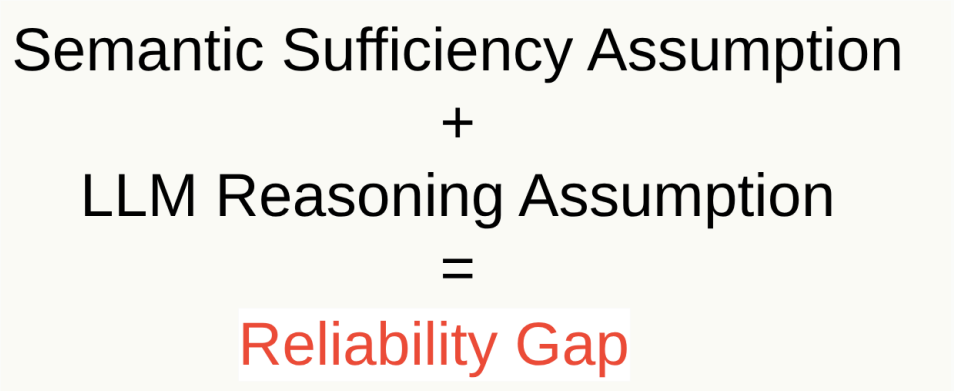
A Practical Guide to Building Reliable AI Systems

v1.0 Feb 2026

In 2026, Retrieval-Augmented Generation (RAG) is the industry baseline for commercial AI. However, as these systems move from controlled pilots into high-consequence production environments, a troublesome **'reliability gap'** often emerges. In high-consequence environments, this gap manifests as silent logical errors rather than obvious hallucinations.

When the standard vector-search-and-summarize pattern collapses under real-world constraints, the root cause is usually related to two precarious assumptions:

1. **The Assumption of Semantic Sufficiency:** The belief that retrieving all information that semantically matches a given query ensures an LLM, given that information, can correctly answer any query.
2. **The Assumption of LLM Reasoning:** The belief that if **all** relevant information is successfully retrieved (even beyond the first assumption) an LLM can synthesize the correct answer.



Daniel
JACOBSEN

Why the assumption of semantic sufficiency is risky

The first assumption is risky because semantics are not always sufficient. For example, say you're building a portfolio intelligence platform. Raw data consists of quarterly earnings, analyst reports, macroeconomic updates, and supply chain disclosures.

Now, one of your users asks: *'How exposed is our portfolio to a potential lithium export ban in Chile?'*

A standard vector search finds sections relating to Lithium, export bans, and Chile. It retrieves a news snippet about the Chilean government's policy. A transcript of a mining company mentioning lithium. But the query requires **connecting the dots** to the rest of the portfolio. It doesn't 'know' that your customer's portfolio owns 4% of a German EV Battery manufacturer that has a sole-source contract with that Chilean mine. How could it - there is no semantic connection between this critical fact and the query. The word 'exposed' may not even be used at all in the raw data.

If we're lucky, the 'naive' RAG system offers an answer like 'Chile is considering a ban. This might affect lithium prices.' The answer is vague and misses the direct hit to the German holdings.

Why the Assumption of LLM Reasoning is Risky

Even if the system is able to retrieve a complete representation of everything in the data that is relevant to the query, the second assumption - that of LLM reasoning - can get us. That's because it fails to account for the lack of robustness in logical reasoning that LLM's suffer from, particularly when operating in specialized domains outside its core training data.

The lack of robustness is the result of three main problems:

1. The Mental Math Problem
2. The Noise Problem
3. The Scale Problem

The Mental Math Problem

Continuing the example above, say our retriever is very smart, and we do not rely on the assumption of semantic sufficiency. Instead of just finding textual snippets, it constructs a graph representation of objects and their relationships - something we will cover when we get to the solutions part of this guide. With this in place, the LLM answer generator gets, as part of its input:

Document Chunk 1 — Portfolio Holdings Report

The portfolio currently includes a 4.2% allocation to Varta AG, a German battery manufacturer specializing in lithium-ion storage solutions. Varta's exposure to upstream lithium supply chains has increased in recent quarters due to long-term sourcing agreements.

Document Chunk 2 — Earnings Call Transcript

Management confirmed that approximately 80% of lithium input materials are sourced from a single Chilean extraction partner operating in the Atacama region.

Document Chunk 3 — Policy News Update

The Chilean government is considering regulatory measures that could restrict lithium exports as early as Q4 2026, pending parliamentary approval.

Document Chunk 4 — Industry Commentary

Analysts warn that export restrictions in Chile could materially impact global lithium pricing and battery manufacturers dependent on South American supply chains.

(here we ignore the significant number of other chunks that relate semantically *but not logically* to the question)

Now we need the LLM to perform 'mental math' to realize the connection. While modern LLMs can do this simple queries, it starts to degrade as the logical path lengthens. We can't know in advance how many "hops" will be needed for new user queries. This is risky - we are hoping that it works. And as we know, hope is a very poor substitute for strategy.

The Noise Problem

Retrieval is rarely perfectly clean. Even a good retriever searching the neighborhood around 'German Battery Corp' might also get content around CEO Names, Stock Tickers, or Office Locations, and so on.

If we naively add everything retrieved into an LLM generator, it has to figure out which edges are relevant to the 'Lithium Ban' and which are just trivia. Again, this can easily fail for realistic queries.

The Scale Problem

This is where the 'just feed everything to the LLM' strategy completely breaks. For example, one user asks:

'Across our entire 500-company portfolio, which sectors are most vulnerable to South American political instability?

The relevant content is now thousands of chunks. You can try to plop this into a context window but the LLM is unlikely to generate the right answer.

In addition to these challenges, naively inserting large amounts of content into LLM context is expensive. Why pay for 20k input tokens if you could manage with 2k?

The Solution: Reasoning Systems Design

When these assumptions fail, the results range from minor embarrassment to outright disaster. One way of trying to fix the problem is brute force: swap the 70B model for a 400B model. Activate (and pay for) 'thinking' mode. Another approach is to do 'more of the same', for example obsessively tweaking the chunking strategy.

These are temporary patches for a structural deficiency. Building systems that are technically defensible and operationally stable requires a shift toward **'Reasoning Systems Design.'**

What is 'Reasoning Systems Design' (RSD)?

This is the discipline of architecting a **system** that is able to **reason** about a specific **domain** so that it gains the capability to answer **domain queries** with a very high degree of **correctness.**

Daniel
JACOBSEN

The Objective

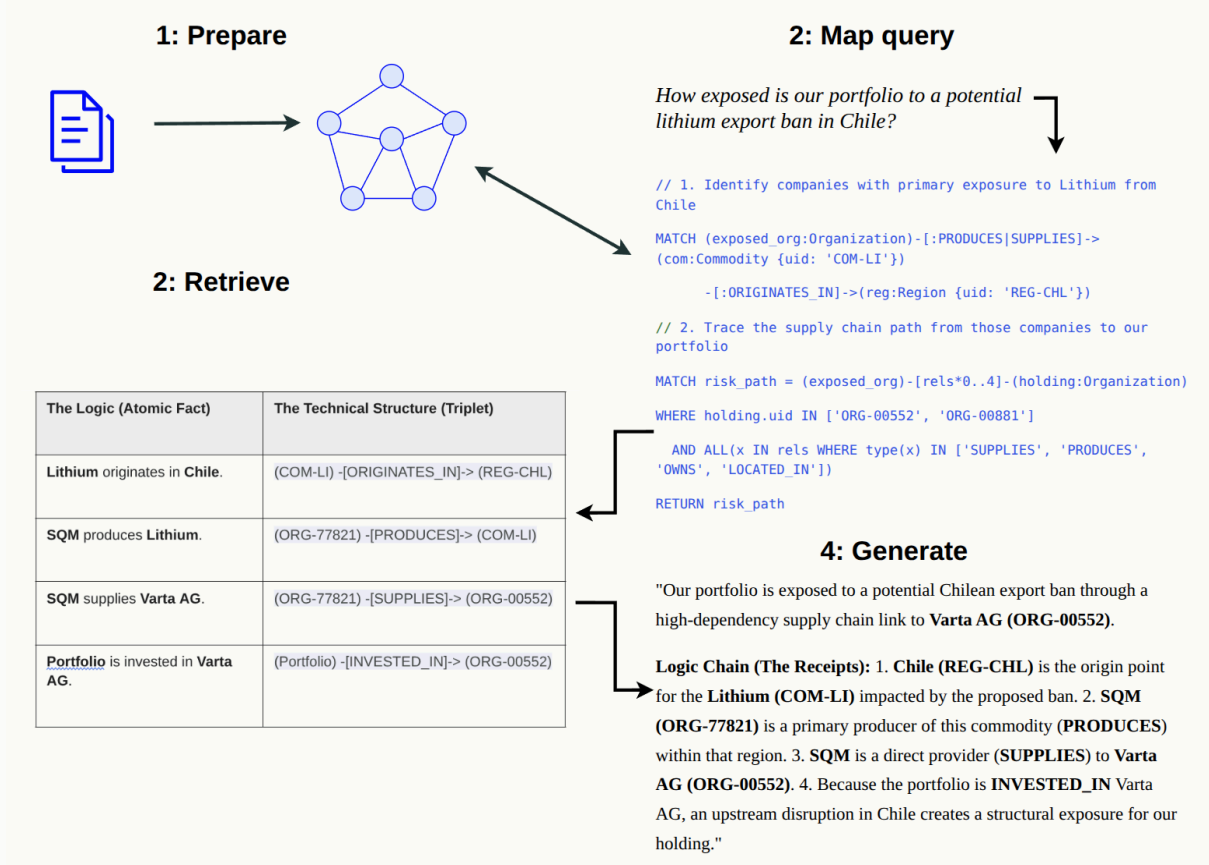
The goal is to provide a **Robust AI System**: one that performs predictably in production environments that are inherently noisy and time-varying.

RSD Phases

A reasoning system follows a deterministic set of phases to maximize reliability:

1. **Preparation:** Transform raw, messy data into a structured knowledge foundation.
2. **Query mapping:** Translate a natural language query into a set of structured retrieval requests matching that structured data.
3. **Retrieval:** Retrieve the relevant structures and properties required to 'solve' the query.
4. **Generation:** Reason over the retrieved structures to arrive at the correct answer, with verifiable reasoning.

The diagram below shows a high-level representation of these phases.



A Minimum Viable Reasoner

Continuing our example from above, the following is a recipe for a simple robust AI system - a Minimum Viable Reasoner (MVR) that you could practically implement for your own domain.

This MVR is a baseline reference architecture. In practice, its boundaries, ontology scope, and traversal depth must be tuned to domain complexity and failure tolerance.

> For relevant design decisions, important **trade-offs** are highlighted like this.

How well this particular MVR performs in your domain depends on a lot of factors - it comes with no warranties. You will definitely have to tweak and even expand it to work well for your real-world challenges. However, it will give you some valuable dirt under your fingernails and get you started on the RSD path - and that is what will take you to the robust AI system that your customers need and that your success depends on.

Phase 1: Preparation

Phase 1 transforms unstructured text into a Domain Knowledge Graph (DKG).

This stage moves beyond semantic indexing to create a 'closed world' of facts where every entity and relationship is anchored to a unique, immutable identifier.

Architecturally, this establishes the structural substrate on which future AI features will depend.

The Ontology Build

Before a single line of code is written, domain experts must first define the ontology. This is the finite set of the things - entities - that exist in your domain, and their relationships. Think of the ontology as the schema and the DKG as the instance, the actual data that must obey the schema.

Engineers and Subject Matter Experts (SMEs) define the core Object Classes and Relationship Classes (Predicates) that define the domain ontology.

Important constraint: Any information in a document that does not fit this schema is ignored. This is a feature, not a bug, because it filters out the noise that causes reasoning failures.

Continuing our example from earlier, our ontology would be:

Nodes: Organization, Asset, Commodity, Region, Event.

Relationships (edges): OWNS, SUPPLIES, PRODUCES, LOCATED_IN, CONSUMES.

> **Trade-off:** Constraining the ontology improves reliability but reduces recall surface.

A Note on Ontology Versioning and Migration Strategy

The ontology is not static. As your product evolves, new object classes and relationships will emerge, and existing predicates may need refinement. Treat the ontology as a versioned artifact with explicit change control.

Each ontology revision should:

- Be version-tagged and stored in source control.
- Define forward migration rules for existing graph data.
- Avoid silent schema drift by blocking ingestion against undefined node or edge types.

Without versioning discipline, incremental ontology changes can introduce inconsistent reasoning paths and invalidate previously deterministic behavior. A reasoning system is only as stable as the evolution strategy of its schema boundary.

Build the Global Entity Index

The GEI is the 'Master Data' anchor. It is a curated, internal SQL database (postgres) populated with known entities before ingestion begins. It serves as the single source of truth to **prevent entity duplication**.

This is simply an SQL table with three mandatory columns: UID (Primary Key), Canonical_Name, and Aliases (a JSONB array of naming variations). For our example, it would look like this:

UID	Canonical_Name	Aliases (JSONB Array)
ORG-77821	Sociedad Química y Minera de Chile	["SQM", "Sociedad Quimica", "SQM-B", "Chilen-Lithium Corp"]
ORG-00552	Varta AG	["Varta", "Varta Storage", "VARTA AG", "VAR1.DE"]

AST-00990	Salar de Atacama Mine	["Atacama Mine", "Salar de Atacama", "Atacama Lithium Project"]
COM-LI	Lithium	["Li", "Lithium Carbonate", "Lithium Hydroxide"]
REG-CHL	Chile	["Republic of Chile", "Chilean Government", "CL"]

The construction of the GEI is a data engineering task that ensures every logical path in your graph has a stable starting point.

Aggregate the GEI from three specific sources:

- The Portfolio Master: Your internal 'golden record' of the entities you care about, such as your list of 2,000 invested companies.
- The Industry Authority: Public datasets that provide immutable IDs. For our example, it might include the S&P Global Commodity Index.
- An Alias Scraper: Use a source like the Wikipedia API script to pull known common names and parent company variations for your alias list.

The GEI becomes the **identity boundary** of the reasoning system. Without it, entity drift reintroduces probabilistic ambiguity into what should be a deterministic structure.

The SQL Schema

Implement the GEI as a Postgres table with a GIN index on the Aliases column to allow for high-speed "array-contains" lookups.

```sql
CREATE TABLE global_entity_index (
    uid VARCHAR(50) PRIMARY KEY, -- e.g., ORG-77821
    canonical_name TEXT NOT NULL, -- e.g., Sociedad Química y Minera de Chile
    entity_type VARCHAR(50),      -- e.g., Organization
    aliases JSONB NOT NULL        -- e.g., ["SQM", "SQM-B", "Sociedad Quimica"]
);
-- Pro tip: Index the aliases for Pass 2 resolution speed
CREATE INDEX idx_gei_aliases ON global_entity_index USING GIN (aliases);
```

Construction Workflow

1. Extract: Pull your golden list of companies and assets from your internal CRM or Portfolio Management System.
2. Enrich: For each entity, call an external API (like S&P) to fetch all legal names and variations.
3. Populate: Insert these into the SQL table.
4. Sanitize: Run a deduplication script to ensure no two UIDs share an alias.

Extracting the DKG

We need a graph database because reasoning requires recursive traversals that are prohibitive in SQL. For the MVR, we use Neo4j because its massive ecosystem of construction tools and deep integrations with LangChain allow

you to move from raw text to a functional graph-connected chatbot faster than most other platforms.

Pass 1: Parallelized Segment Extraction

Use a Sliding Window to manage a large number of documents.

- Segmentation: Documents are broken into fixed 2,000-token segments with a 400-token (20%) overlap to capture relationships spanning page breaks.
- Extraction: A high-reasoning LLM, Claude Sonnet 4.5) extracts raw triplets.
- The Tech: Instructor uses Pydantic models to force the LLM to return a validated list of objects. If the LLM returns a malformed string, the Instructor library automatically retries the call.
- The Prompt: "Extract relationships from the text using ONLY the DKG Schema. Format your output to match the TripletList Pydantic model."
- The Sink: The validated Pydantic objects are serialized and written to a Postgres Staging Table. Each row contains the raw subject, predicate, object, and a reference to the source document chunk.

Pass 2: Canonical Resolution & Commitment

A Python middleware script intercepts the raw staging data and performs an exact alias match against the GEI. For each sub and obj, the script queries the SQL Aliases array in the GEI. If, say, 'Varta AG' matches an alias for UID:ORG-552, the script overwrites the string with that UID.

The resolved, deduplicated triplets are then written to Neo4j. Identical facts from multiple sources collapse into a single, high-confidence edge.

Note that If the LLM extracts "Salar de Atacama", the script doesn't ask the LLM "What is this?" It simply runs:

```sql
SELECT uid FROM global_entity_index WHERE aliases @> '"Salar de Atacama"';
```

It receives, say, “AST-00990” and writes that ID to Neo4j. This is how you achieve deterministic entity resolution within the defined ontology boundary.

Implementation Checklist

Component	Responsibility	Technical Requirement
Ontology Build	SME + AI Lead	Defined list of Node/Edge types.
GEI Database	Data Engineer	Postgres table with indexed JSONB for Aliases.
Extraction Pipeline	AI Engineer	Parallel async script using the Python Instructor library.
Resolution Script	Backend Engineer	Python middleware for SQL lookup and Neo4j commitment.

Phase 2: Query Mapping

Phase 2 is the "Compiler" stage. We never allow a raw, messy user question to hit the graph database directly. Instead, we translate the natural language intent into a formal logical execution plan. This ensures the reasoning follows the strict paths defined in your ontology.

The ontology becomes the boundary between probabilistic extraction and deterministic reasoning. Its stability determines long-term system behavior.

Entity extraction

First, we must find all the relevant entities. To do that, we use a **tool-calling AI agent** that has access to three specific functions:

- **resolve_external_entity(term):** Queries the GEI to map user words (e.g., "Chile") to UIDs (REG-CHL).
- **fetch_portfolio_uids():** Queries your Internal SQL DB of holdings, runs them through the GEI, and returns the list of UIDs currently in your "basket."
- **plan_path(start_uid, target_uids):** Uses the Ontology Schema to determine which predicates (like SUPPLIES) connect the two.

The Path Planner: LLM-as-Compiler

Once the system has resolved the starting point (REG-CHL) and the targets (Portfolio UIDs), the LLM acts as a query compiler. Its job is to map the user's natural language intent to a search plan that aligns with our ontology.

This is something modern LMMs actually are able to do remarkably robustly. Note how we decompose the problem so that instead of expecting the LLM to do all of the reasoning, it now has the much easier task of generating an execution plan. And that in turn allows us to make the next step, that of generating the execution of that plan, much easier.

The JSON Execution Plan

For the query “How exposed is our portfolio to a potential lithium export ban in Chile?”The planner decomposes the problem into a search plan.

Instead of assuming the database "knows" what an export ban is, the planner identifies the **anchors** (Chile, Lithium) and the **targets** (Portfolio UIDs) and maps the logic required to bridge them.

This particular plan identifies that we must find organizations that produce the commodity in the region and then trace a variable-depth path to our holdings. The resulting JSON is a direct blueprint for a graph traversal. Each step represents a required "hop" in the database.

JSON

```
{
  "path_logic": [
    {"from": "Region", "uid": "REG-CHL"},
    {"from": "Commodity", "uid": "COM-LI"},
    {"step": "Identify exposed producers", "logic":
"(Org)-[:PRODUCES]->(COM-LI)-[:ORIGINATES_IN]->(REG-CHL)"},
    {"step": "Trace to portfolio", "edge": "ANY", "depth":
"1..4", "targets": ["ORG-00552", "ORG-00881"]}
  ]
}
```

The Coder LLM: Plan to Path

The input to this LLM is the JSON Execution Plan (from Step 2) and the Ontology (from Phase 1).

The prompt starts with an instruction: "Translate the following Logical Execution Plan into a single, valid Cypher query for Neo4j. Use the provided Ontology to ensure relationship types are exact."

To ensure the LLM handles the ANY step correctly, we give it clear conversion rules:

- Rule for TRAVERSE: Translate to a direct relationship: (a)-[:REL]->(b).
- Rule for ANY: Use the Variable-Length Path Pattern with a schema-bound predicate filter.

We provide examples like:

- *Input Step:* `{"action": "ANY", "depth": "1..3", "to": "Organization", "targets": ["ID-1"]}`
- *Output Cypher:* MATCH p = (prev)-[r*1..3]-(target:Organization) WHERE target.uid IN ['ID-1'] AND ALL(rel IN r WHERE type(rel) IN ['SUPPLIES', 'PRODUCES', 'OWNS', 'LOCATED_IN'])

Trade-off: Variable-depth traversals increase expressiveness but must be bounded to prevent combinatorial explosion.

The Final Assembled Cypher

Given this prompt, the LLM produces a deterministic implementation of the requested logic:

```
None

// 1. Identify companies with primary exposure to Lithium
from Chile

MATCH
(exposed_org:Organization)-[:PRODUCES|SUPPLIES]->(com:Com
modity {uid: 'COM-LI'})

      -[:ORIGINATES_IN]->(reg:Region {uid: 'REG-CHL'})

// 2. Trace the supply chain path from those companies to
our portfolio

MATCH risk_path =
(exposed_org)-[rels*0..4]-(holding:Organization)

WHERE holding.uid IN ['ORG-00552', 'ORG-00881']

  AND ALL(x IN rels WHERE type(x) IN ['SUPPLIES',
'PRODUCES', 'OWNS', 'LOCATED_IN'])

RETURN risk_path
```

Why this plan-code decomposition is important

1. **Isolation of Concerns:** The first LLM focuses only on strategy (e.g., "I need to look for a ban impact on lithium"). The second LLM focuses only on Syntax (e.g., "How do I write a variable-depth path in Cypher?").
2. **Validation**: It is much easier to validate Cypher given a plan and an ontology, than against a raw user question (we do not cover such validation here.)

3. **Traceability**: This separation creates observable failure boundaries, allowing debugging at the logic or syntax layer independently.

If the compiler stage is skipped and user questions hit Cypher directly, ontology discipline collapses over time

Implementation Checklist

Component	Responsibility	Technical Requirement
Entity Resolver Agent	AI Engineer	Tool-calling LLM that maps query terms to **GEI UIDs**.
Portfolio Bridge	Backend Engineer	Logic to pull internal holdings and resolve them to **GEI UIDs**.
Logic Planner	AI Engineer	LLM that generates the step-by-step path logic.
Hybrid Cypher Coder	AI Engineer	LLM that writes the Cypher with variable-depth `[*1..4]` hops.
Semantic Guardrail (optional)	Backend Engineer	Python tool to block any non-ontology relationships/classes.

Phase 3: Retrieval (The Structural Harvest)

Now we simply execute the Cypher query from phase 2 against the Domain Knowledge Graph (DKG) that we built in phase 1.

Executing the Path Search

Our system takes the Cypher code and runs it against our Neo4j graph database. Because we used the variable-depth [*1..4] logic, the database isn't just looking for words; it is mathematically identifying the logical "roads" that connect the Chilean Ban to specific, relevant Portfolio UIDs.

Capturing the "Logical Neighborhood"

Instead of a wall of text, the output of Phase 3 is a set of path objects. This is the specific sub-section of your graph that proves exposure.

- Input: A validated Cypher query.
- Action: The Neo4j engine traverses the relationships (PRODUCES, SUPPLIES, OWNS).
- Output: The exact chain of entities (Nodes) and relationships (Edges) that link the "trigger" to the relevant holdings.

The Path Parser: From Graph to Logic

Neo4j returns path objects. These objects are rich in database metadata that would confuse an LLM (such as internal node IDs and property hashes). To solve this, a deterministic **Python Path Parser** follows a strict three-step logic:

- **Unpacking Segments:** It iterates through the `segments` of the retrieved paths. Each segment represents one "hop" in the graph (e.g., from a Commodity to an Organization).
- **Canonical Mapping:** For every node in a segment, the parser ignores internal database IDs and extracts the **Canonical Name** and **UID** that were anchored during Phase 1 Preparation.
- **Triplet Assembly:** It reformats each segment into a clean **Triplet** (Subject-Predicate-Object).

Output: Atomic Facts

The result of this parsing is a set of **Atomic Facts:** the specific semantic links that "pre-solve" the query for the final reasoning step. For our Chile query, the output looks like this:

The Logic (Atomic Fact)	The Technical Structure (Triplet)
Lithium originates in **Chile**.	(COM-LI) -[ORIGINATES_IN]-> (REG-CHL)
SQM produces **Lithium**.	(ORG-77821) -[PRODUCES]-> (COM-LI)
SQM supplies **Varta AG**.	(ORG-77821) -[SUPPLIES]-> (ORG-00552)
Portfolio is invested in **Varta AG**.	(Portfolio) -[INVESTED_IN]-> (ORG-00552)

Both the natural-language version (first column) and the triplets (second column) are given to the LLM that will generate the final answer.

Daniel
JACOBSEN

Why this solves the "Mental Math Problem"

By retrieving the logical paths, we stop relying on the "Assumption of LLM Reasoning". The LLM doesn't have to guess or connect the dots in its “head” across multiple hops; the retrieval phase has **already connected** them for it. We are handing the LLM a "pre-solved" logical puzzle.

Implementation Checklist

Component	Responsibility	Technical Requirement
Graph Executor	Backend Engineer	A driver (e.g., `neo4j-python-driver`) to run the validated Cypher.
Path Parser	Data Engineer	Logic to unpack Neo4j "Path" objects into a simple list of triplets.
Result Bundle	Data Engineer	A script to package these paths into a structured format for the final reasoning stage.

Phase 4: Answer Mapping (The Narrated Receipt)

Phase 4 is the stage where the system translates retrieved structures and properties into the final response. In the Minimum Viable Reasoner (MVR) architecture, this is primarily an Answer Mapping exercise. Rather than relying on the "Assumption of LLM Reasoning" to guess connections from text snippets, the system uses an LLM to narrate the pre-solved logical path extracted from the graph database during Phase 3.

The Mapping Input: The "Fact Pack"

The final LLM receives the user's original query and the Fact Pack generated during the retrieval phase. This pack consists of the "Atomic Fact" triplets along with their unique, immutable identifiers (UIDs) from the Global Entity Index (GEI).

Including UIDs (e.g., ORG-77821) in this phase is essential for several reasons:

- **Enforced Grounding**: It locks the LLM into a "closed world" of facts, preventing it from introducing outside "noise" or training-data trivia.
- **Verifiable Reasoning**: It allows the final answer to include "receipts", which are verifiable citations that point directly back to the structured knowledge foundation.
- **Solving the Scale Problem**: By feeding the LLM only the few specific facts that constitute the identified risk path, we avoid the context window bloat and excessive costs associated with feeding the kitchen sink to the LLM.

The LLM performs a kind of "narrative mapping" of the structural data. Because the structural "mental math" - identifying the multi-hop connection between the Chilean lithium and your portfolio - was already executed by the graph traversal in Phase 3, the LLM's role is to describe that path accurately and attach the UIDs as verifiable evidence. While some level of inference is still required, this task is much simpler than the whole query-to-answer one inference task.

The Final MVR Response

With this, we get a correct, verifiable, answer:

"Our portfolio is exposed to a potential Chilean export ban through a high-dependency supply chain link to **Varta AG (ORG-00552)**.

Logic Chain (The Receipts): 1. **Chile (REG-CHL)** is the origin point for the **Lithium (COM-LI)** impacted by the proposed ban. 2. **SQM (ORG-77821)** is a primary producer of this commodity (**PRODUCES**) within that region. 3. **SQM** is a direct provider (**SUPPLIES**) to **Varta AG (ORG-00552)**. 4. Because the portfolio is **INVESTED_IN** Varta AG, an upstream disruption in Chile creates a structural exposure for our holding."

Implementation Checklist

Component	Responsibility	Technical Requirement
The Reasoner	AI Engineer	High-reasoning LLM (Claude 4.5 Sonnet) prompted with the structured **fact pack**.
Citation Guardrail	AI Engineer	Prompt instructions that mandate the inclusion of UIDs as "receipts" for every claim made.
Abstention Logic	Backend Engineer	A deterministic protocol for the LLM to "politely abstain" if the retrieved facts do not support a complete logical path.

Daniel
JACOBSEN

Where to go from here

Congratulations on making it this far.

Reasoning Systems Design (RSD) is an emerging engineering discipline, and it is evolving rapidly. The reference architecture outlined in this guide is a deliberate simplification - a functional baseline for moving beyond the inherent limitations of the standard vector-search-and-summarize pattern.

By implementing this architecture, you have begun to address the core Reliability Gap that plagues production AI. You have shifted from a reliance on the Assumption of Semantic Sufficiency and the Assumption of LLM Reasoning toward a technically defensible system that prioritizes structural truth over statistical probability.

Beyond the MVR

While the MVR provides a starting point, building for high-consequence environments at scale introduces further complexities. Future iterations of your Reasoning System may need to address a wide range of aspects like:

- Query Decomposition: Orchestrating sub-queries to handle broad, cross-sector analysis where the relevant graph may exceed thousands of nodes.
- Hybrid Retrieval: Integrating semantic vector search as a secondary, complementary tool for unstructured nuances while maintaining the Knowledge Graph as the primary source of truth.
- Cost & Latency Optimization: Implementing more sophisticated caching and pruning strategies to maintain high reasoning quality while minimizing token consumption.

If This Is Mission-Critical

I work with CTOs and engineering leaders responsible for AI systems that must perform reliably under real-world constraints. My role is to help you design reasoning architectures that teams can implement with confidence and operate without structural blind spots.

I take on a limited number of architecture engagements. If this is core infrastructure for your product, you can book a time here.

Daniel
JACOBSEN

Resources

Excellent introduction to knowledge graphs

Probably the best single overview paper about graph retrieval