Understanding and Using RAG

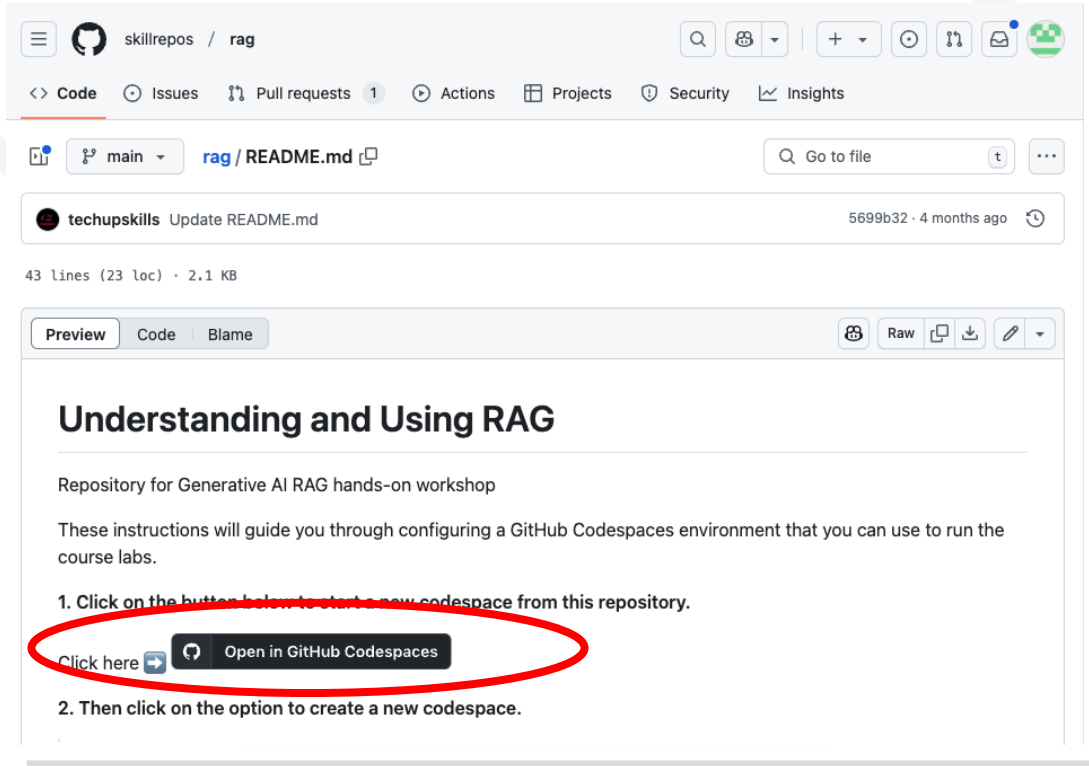
Making LLMs Smarter by pairing your data with Gen AI

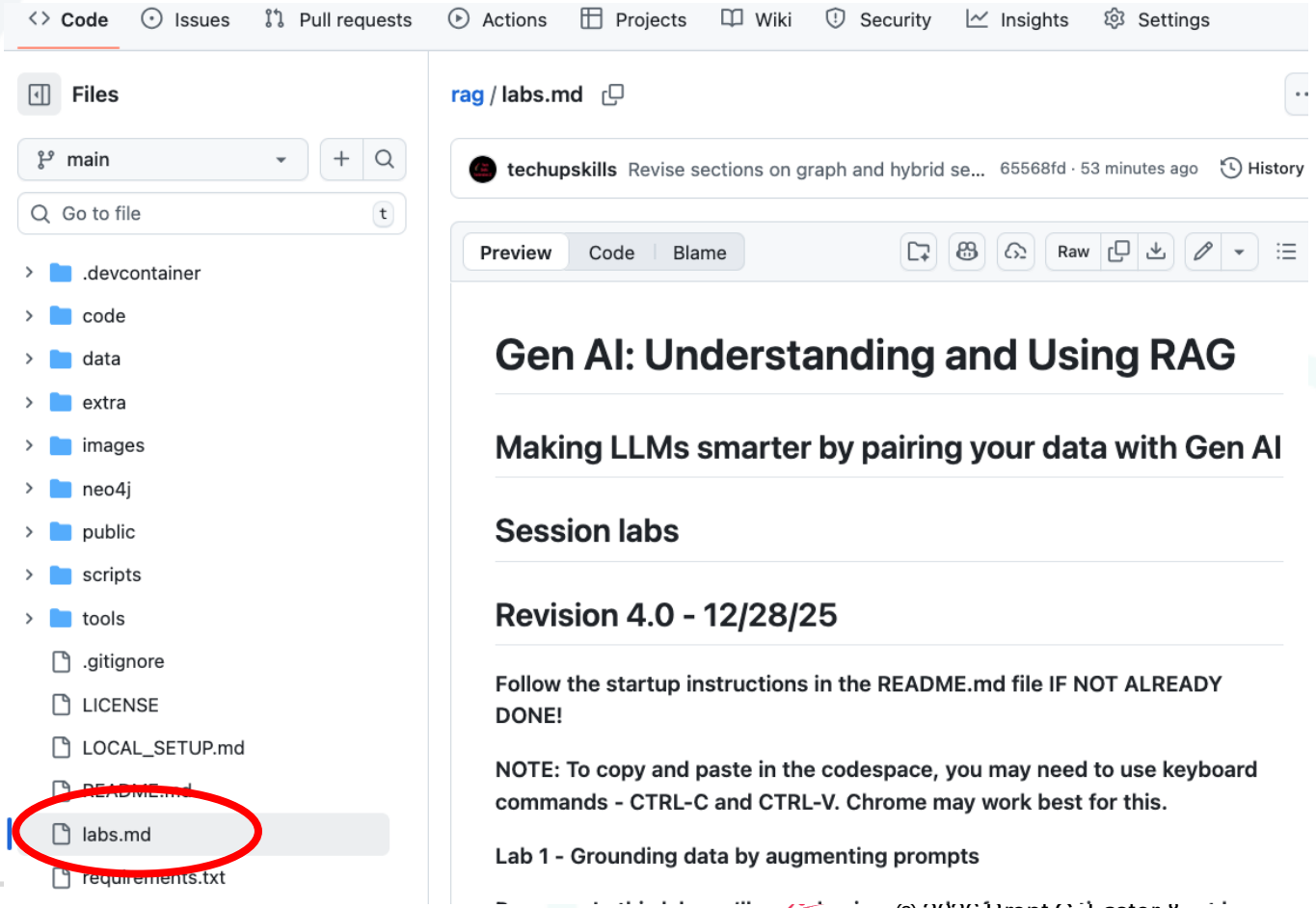
Presented by Brent Laster &

Tech Skills Transformations LLC

Lab prep – repo is github.com/skillrepos/rag

- Go to https://github.com/skillrepos/rag (Chrome may work best for copy and paste actions.)
- Follow instructions in **README.md**
- Startup codespace with quickstart button in README . (*This will take a while to run!)*
- Open *labs.md* in codespace -or- in web



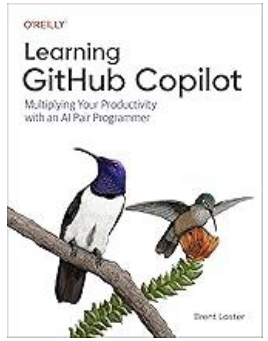


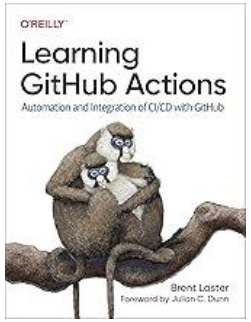
About me

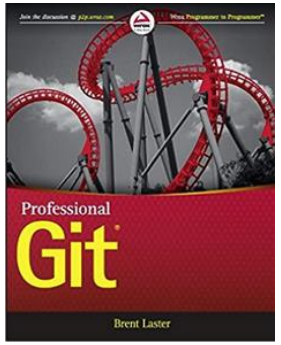
- LinkedIn: brentlaster
- X: @BrentCLaster
- Bluesky: brentclaster.bsky.social
- GitHub: brentlaster

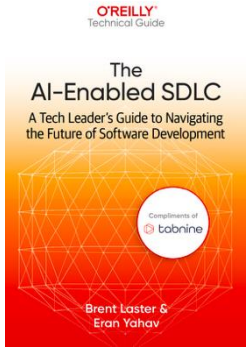
Long career in corporate:

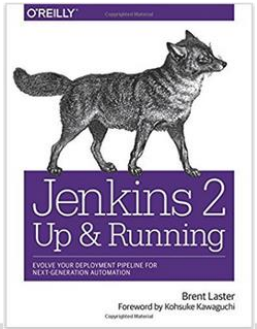
- *Principal Dev*
- *Manager/Senior Manager*
- *Director*











- Founder, Tech Skills Transformations LLC
- https://getskillsnow.com
- info@getskillsnow.com

With Tech Skills Transformations, you don't have to imagine. With new AI training on agents, MCP, RAG, LLMs, and traditional DevOps training from Git to Kubernetes, we provide the understanding, skill development, and productivity you've been looking for. Every workshop incorporates hands-on experiences to help you build confidence, proficiency, while learning how the tech works and applies to you. At Tech Skills Transformations, your success, understanding, and growth is our goal.

Connect: LinkedIn • Web: getskillsnow.com • Email: info@getskillsnow.com

Learn More »

Logistics

- 90-Minutes Training
- 30-Minutes Break
- 90-Minutes Training
- Hands-on labs with time to complete each lab

Agenda – what we'll cover

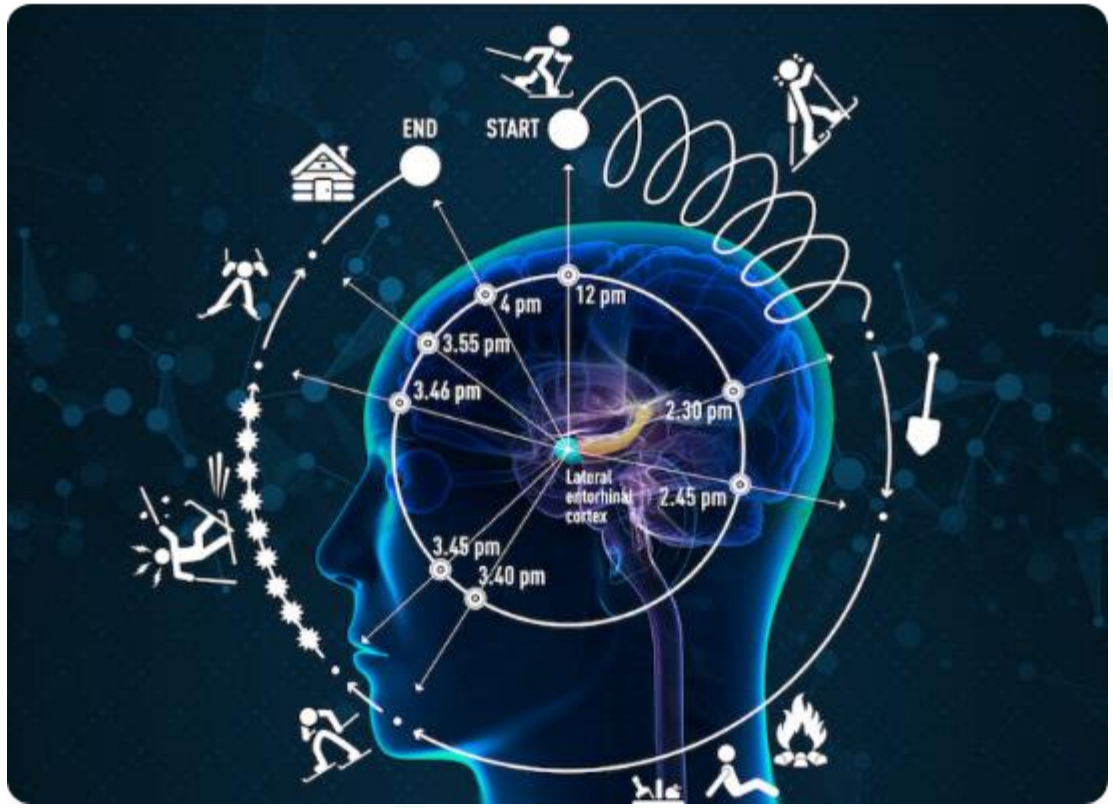
- What is RAG?
- How does RAG work?
- Ollama
- Data stores for RAG
- Embeddings and Vectors
- Vector databases
- Chunking
- Data ingestion pipeline
- Parsing data for RAG
- Full RAG implementations
- Prompting Templates
- Knowledge Graphs
- Graph RAG
- Neo4j and Cypher
- Data stores for RAG
- Frameworks for implementing RAG
- Hybrid RAG

Introduction to RAG

The Knowledge Problem

What LLMs Don't Know

- **Knowledge Cutoff:** Training stopped at a specific date in the past.
- **Proprietary Data:** Private company manuals or internal documents.
- **Real-time Events:** News or data updates happening right now.
- **Specific Procedures:** Unique internal troubleshooting guides.



The Problem Illustrated

User Prompt

"How long do I hold the power button to force restart an OmniTech device?"

Standard LLM Response

"Typically 5-10 seconds for most electronic devices, though it varies by manufacturer..."

Generic Guess

The LLM has never seen OmniTech's internal manual, so it relies on generic probabilities.

Possible Solutions

Approach	Pros	Cons
Fine-tuning	Permanent knowledge absorption	Expensive, slow, requires deep expertise
Retraining	Full structural integration	Extreme cost, impractical for dynamic data
RAG	**Fast, cheap, dynamic**	Requires a separate retrieval system

Winner: RAG – Update knowledge in milliseconds, not months.

What is RAG?

Retrieve + Augment + Generate

An "Open-Book Exam"

Memorization vs. Reference

Without RAG: Like taking a test from memory. You can only answer based on what you studied months ago.

With RAG: You get to bring your notes to the exam. You look up specific facts when needed and combine them with your existing knowledge.

> RAG gives the LLM "cheat sheets" for every question it receives.

How RAG Works Step-by-Step

1. Retrieve

Search your documents for content related to the user's question.

2. Augment

Add that found information into the prompt as "Context".

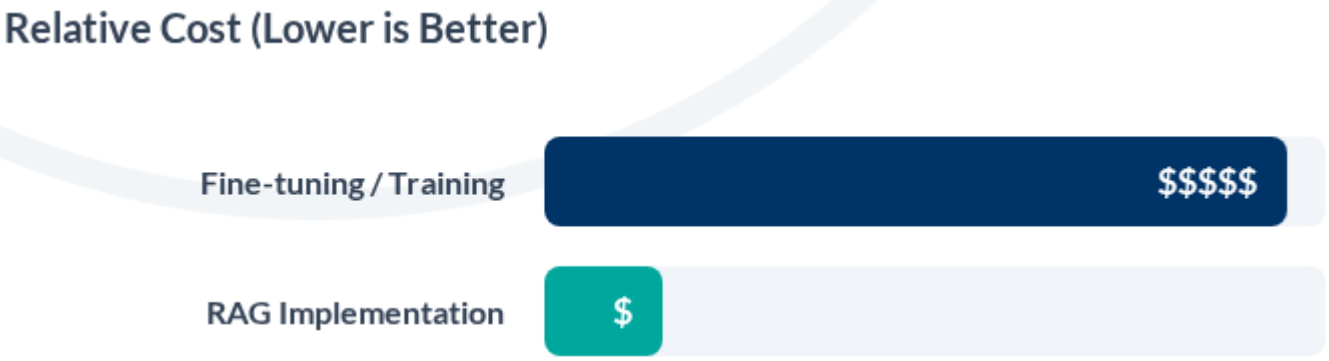
3. Generate

The LLM reads the context and answers based on facts, not guesses.

Before vs. After RAG

Feature	Without RAG	With RAG
Knowledge	Training data only	Training + **YOUR** data
Accuracy	Generic / Hallucination risk	Specific / Grounded in facts
Prompt	Question only	Context + Question
Source	Unknown origin	**Traceable** to docs

RAG vs. Training: Economics



Training is slow and static. If your data changes, you must pay to retrain.

RAG is instant and dynamic. Change the document in your folder, and the AI is updated immediately.

The RAG Mindset

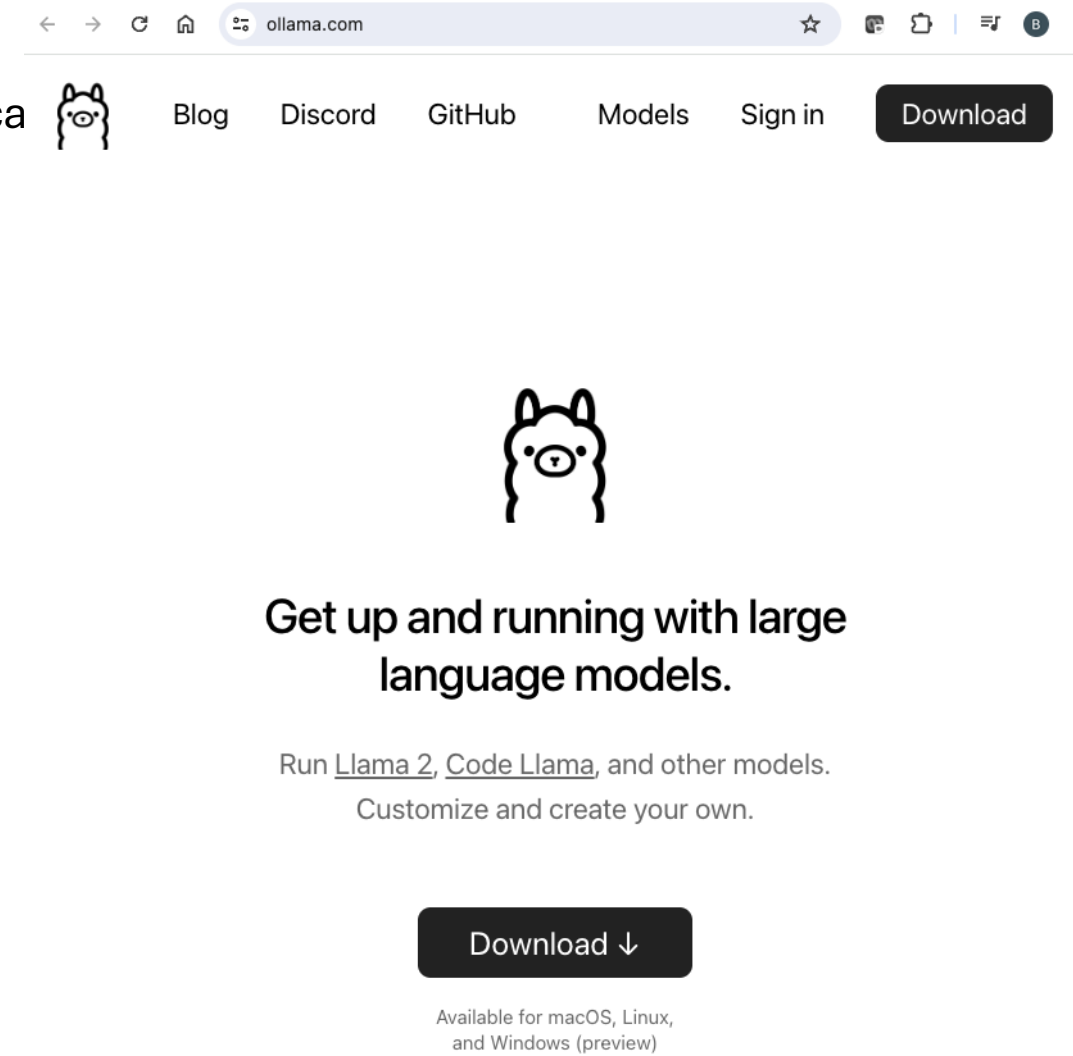
"Don't change the model.
Change what you GIVE the model."

RAG works with any LLM, updates instantly, and grounds AI in your traceable data.

Working with Local LLMs

Ollama

- Command line tool for downloading, exploring and using LLMs on loca machine
- open source
- supports most of Hugging Face's popular models
- allows uploading new ones
- Links:
  - main site: https://ollama.com
  - GitHub: https://github.com/ollama/
- Advantages
  - speeds up and simplifies
    - » model selection and download
    - » configuring endpoints
    - » integration with Python or JavaScript codebase



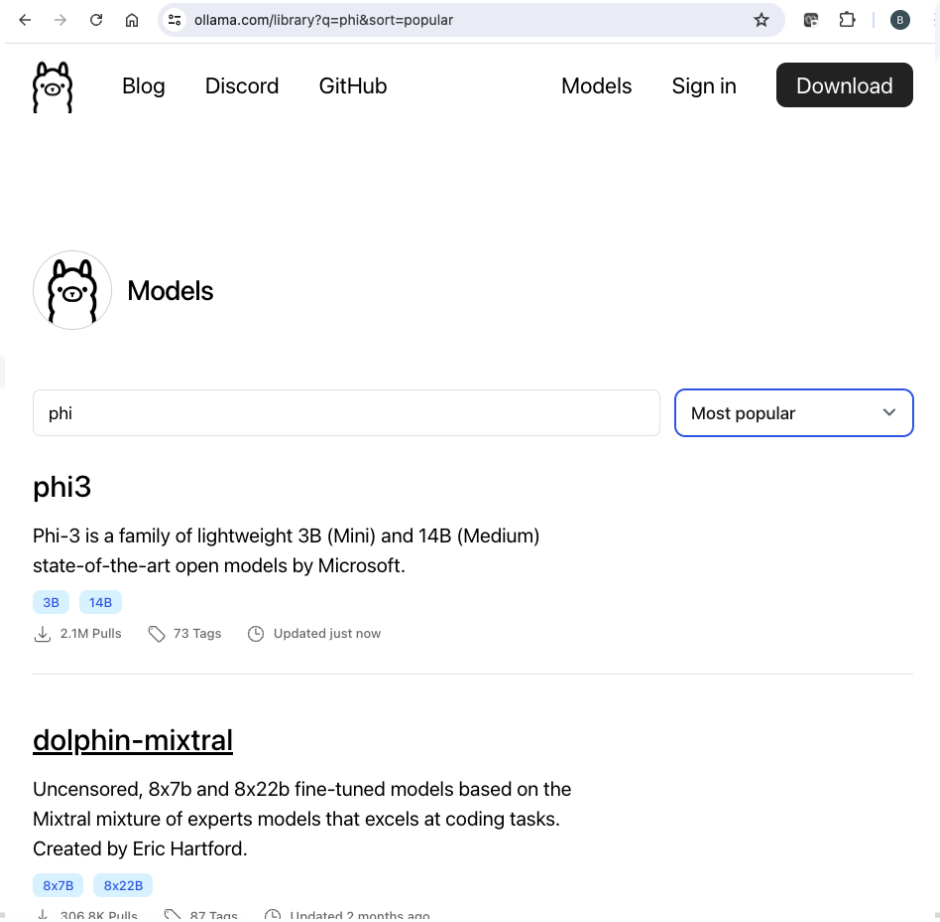
Functions of Ollama

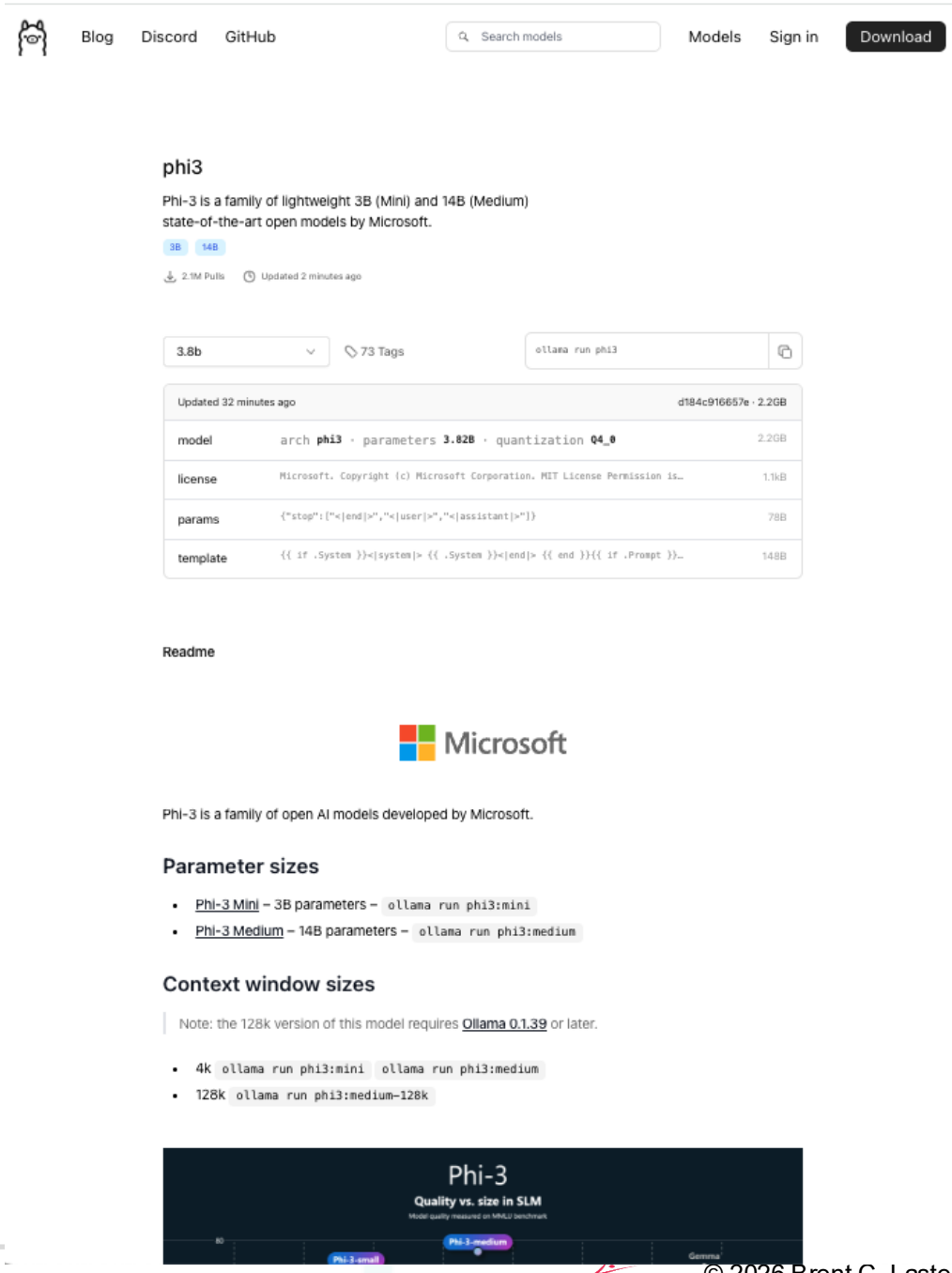
- Typically used functions
  - *serve:* starts ollama
  - *show:* shows info about a specific model
  - *run:* allows you to run a previously downloaded model
    - If model is not present, ollama will download it
  - *pull:* downloads a model, without running it once finished
  - list: prints the list of models available on the machine on the screen
  - *rm:* removes the model

```
● @techupskills → /workspaces/diy-gen-ai (main) $ ollama
Usage:
  ollama [flags]
  ollama [command]

Available Commands:
  serve       Start ollama
  create      Create a model from a Modelfile
  show        Show information for a model
  run         Run a model
  pull        Pull a model from a registry
  push        Push a model to a registry
  list        List models
  ps          List running models
  cp          Copy a model
  rm          Remove a model
  help        Help about any command

Flags:
  -h, --help      help for ollama
  -v, --version   Show version information

Use "ollama [command] --help" for more information about a command.
```

Ollama Models

- Not all models are supported
- To find supported ones, go to https://ollama.com/library
- Click on model link to learn more





Lab 1 Preview: Manual RAG

Step 1: Ask a question about fictional "OmniTech" and observe a generic guess.

Step 2: Open the OmniTech documentation file provided in your environment.

Step 3: Manually copy documentation text into your LLM prompt.

Step 4: Ask the same question and observe the exact, factual response.

Goal: Understand the power of "Augmenting" the prompt with external data.

Lab 1 – Grounding data by augmenting prompts

Purpose: *In this lab, we'll see a basic example of augmenting a prompt by retrieving context from a data file*

Where We're Headed (Labs 1-6)

Lab 1

Manual RAG & Prompt Injection

Lab 2-3

Vector DBs & Automated Systems

Lab 4-5

Knowledge Graphs & Relationships

Lab 6

Hybrid RAG: The Final System

Preparing Your Data

Chunking

Breaking long documents into small, manageable pieces so the AI can find specific facts easily.

Embedding

Converting text into numerical vectors (lists of numbers) that represent the mathematical "meaning" of the words.

Indexing

Storing these vectors in a specialized "Vector Database" for lightning-fast retrieval later on.

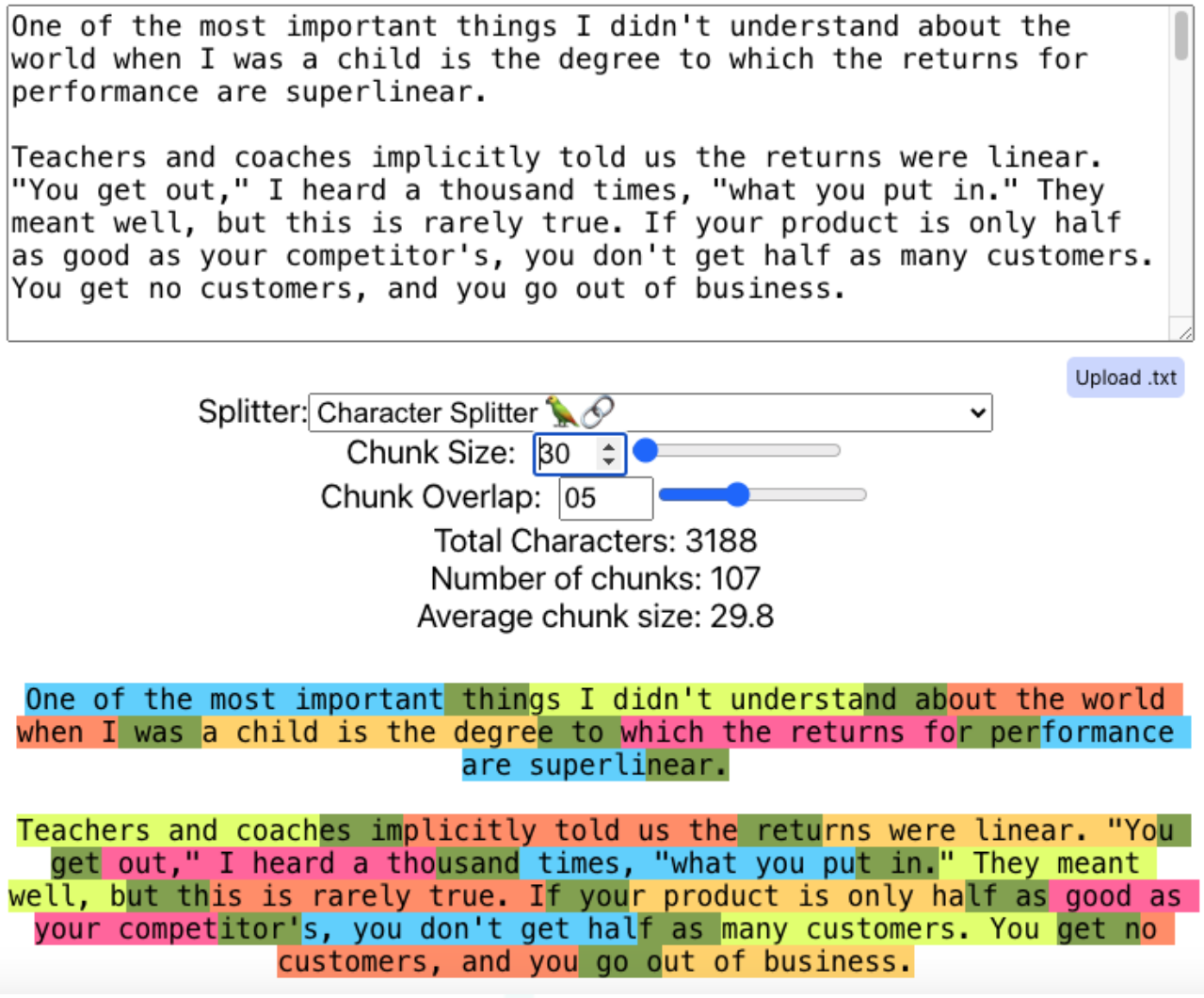
Chunking

Chunking in RAG

- Process of breaking down large documents or pieces of text into smaller, more manageable segments ("chunks")
- Generate embedding for each chunk (e.g. OpenAI embeddings, sentence_transformer) using an encoder and store it in a database.
- Allows for more efficient search and getting relevant information
- Find the Top-K most similar encoded chunks, get the raw text of those chunks, and feed it as context alongside the prompt to the generator.

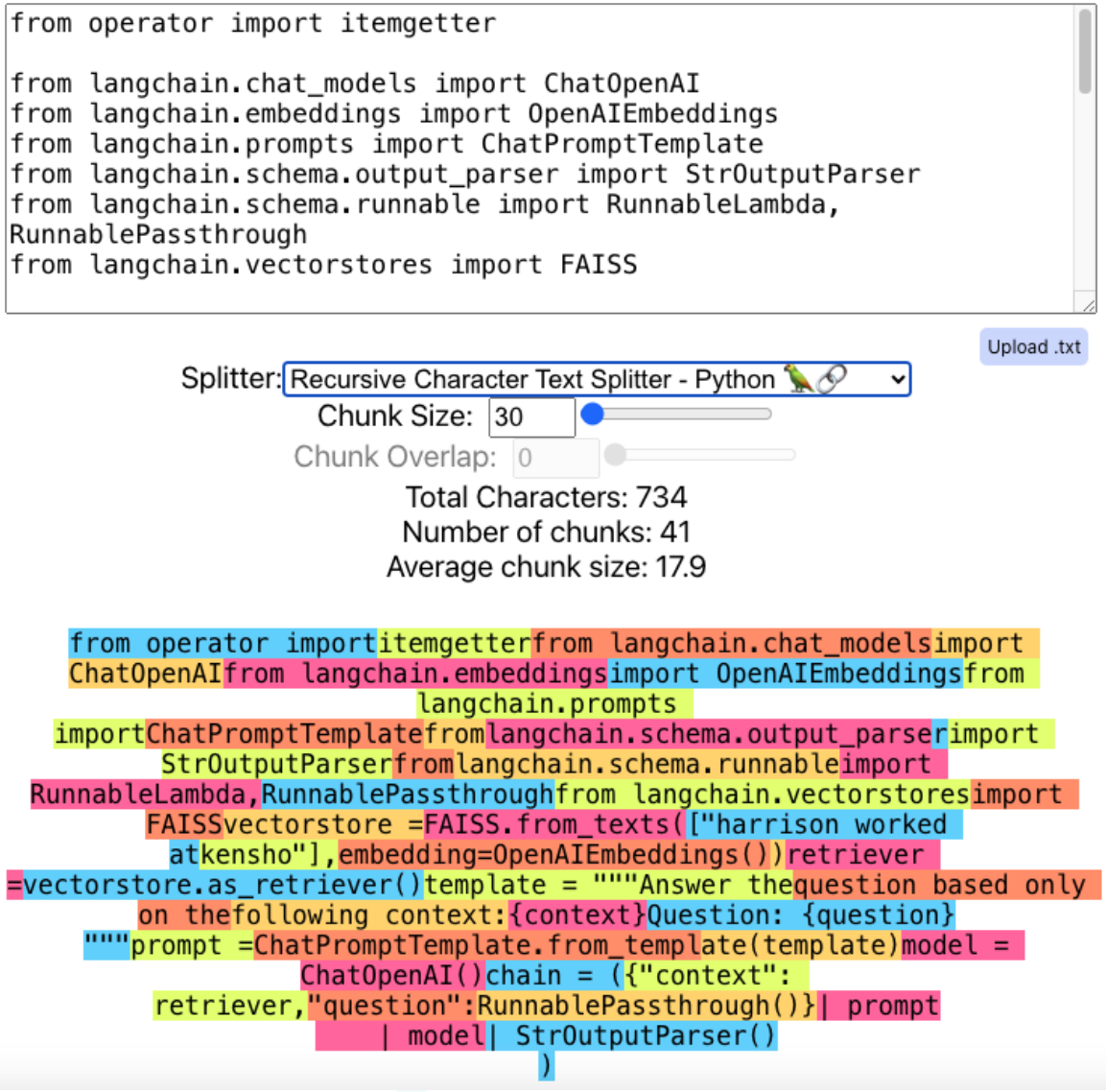
Simple Chunking Strategies for RAG

- Fixed size
  - Works well for content with similar sizes and formats (i.e. articles)
  - Least requirement for compute
  - Doesn't consider any context of content
- Random chunk size
  - Useful for dissimilar collection of content (i.e. multiple document types)
  - May be able to capture more "semantics" w/o depending on structure of any one doc type
  - Risky because might end up splitting into meaningless chunks
- Both use chunk overlap
  - applies chunking over sliding windows
  - new chunks overlap content of previous one; contain part of it
  - allows for better capturing context around edges of chunks and increases semantic relevance overall



Advanced Chunking Strategies for RAG

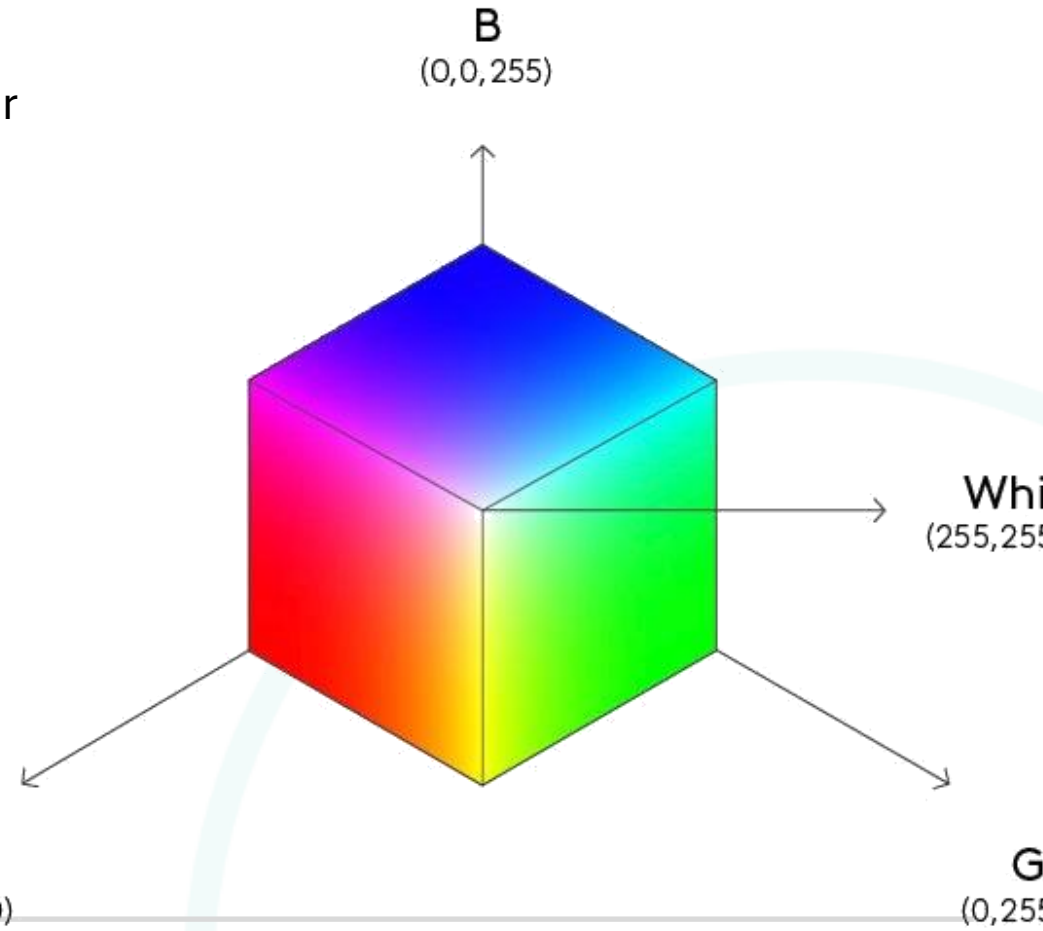
- Simple methods don't work for some kinds of content
  - Example: code
  - Simple methods would result in broken code
- Context-aware
  - Splits docs based on punctuation
  - Examples: commas, periods, paragraph breaks, markdown tags, HTML tags
  - Can recursively chunk docs into smaller pieces that overlap
    - » Chapter gets vectorized/linked as well as each page, paragraph and sentence
  - Requires pre-processing to segment text - can slow down process
- Adaptive
  - Builds on *context-aware*
  - Uses machine learning to figure out best size and overlap for any chunk
  - More complex and more overhead
  - Produces highly tailored semantic units



Embeddings and Vectors

Embeddings

- Embeddings represent text as sets of numeric data - tensors (lots of dimensions)
- Each dimension stores some info about the text's meaning, context, or syntactical aspects
- Words or sentences with similar meanings are stored closer together in the vector space
  - If two pieces of text are similar syntactically, they will have similar embeddings (smaller distance between their vectors)
- During training, models learn to place text with similar meanings closer together in the embedding space
- Common pre-trained models used for generating embeddings include BERT and variants (RoBERTa, DistilBERT)
- Once you have embeddings, you can use them for NLP tasks like semantic search, text classification, sentiment analysis



Semantic meaning / relationships

- Suppose we have 3 words
- King and Queen are more similar to each other than they are to lunch
- In order for neural net to understand the relationships, each word needs to be represented as a vector
- Suppose each word is represented by a 2-dimensional vector

King $\begin{bmatrix} -130.16 \\ 89.5 \end{bmatrix}$

Queen $\begin{bmatrix} -115.43 \\ 95.2 \end{bmatrix}$

Lunch $\begin{bmatrix} -89.5 \\ 34.3 \end{bmatrix}$

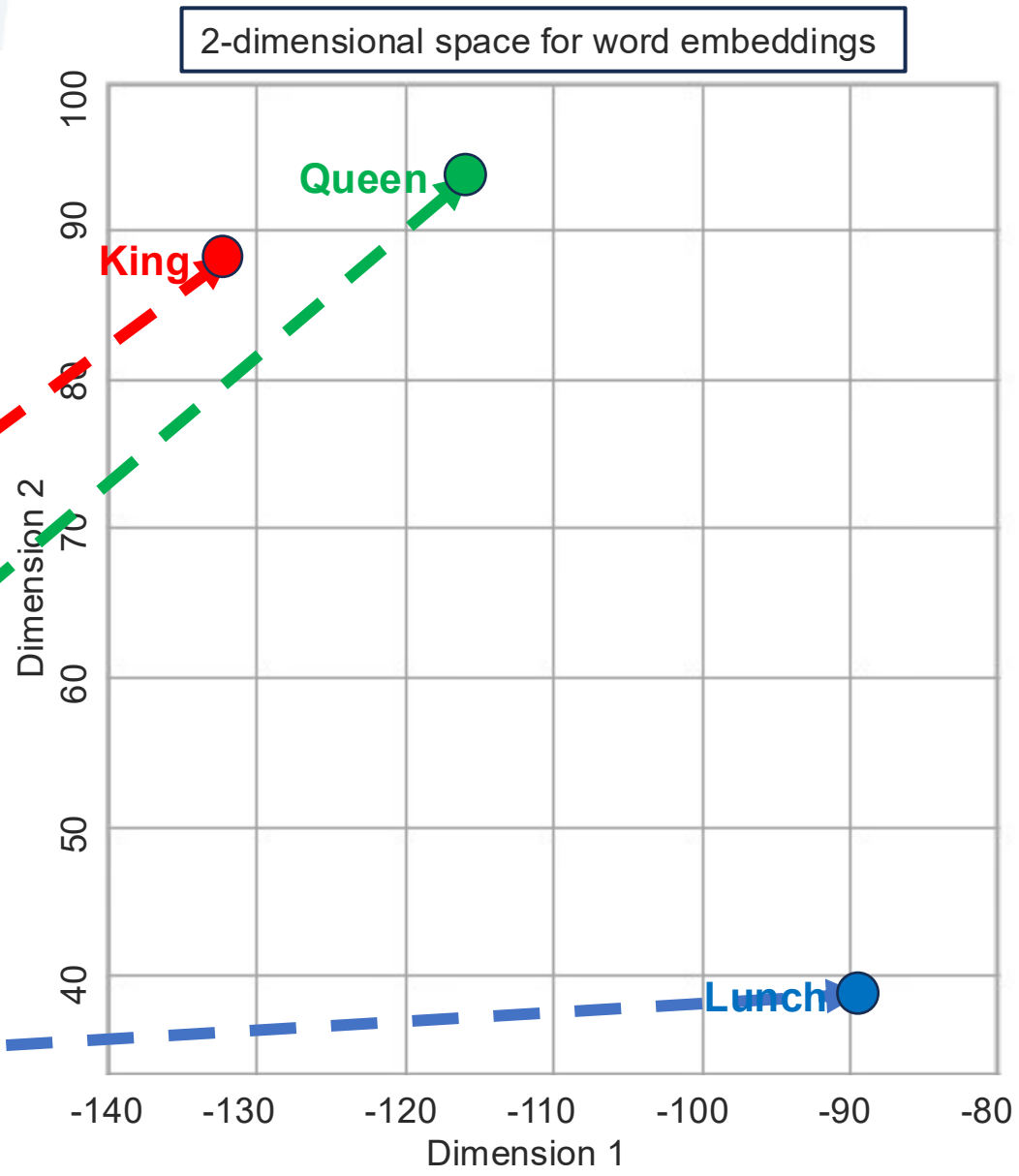
Embedding space

- Plotting in 2-dimensional embedding space shows relationships
- Way to let NN understand relationships between words
- We want the NN to learn that King and Queen are more similar to each other than they are to lunch

King [-130.16, 89.5]

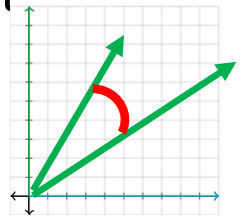
Queen [-115.43, 95.2]

Lunch [-89.5, 34.3]

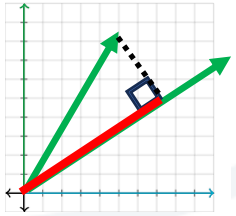


Searching for Vectors – similarity metrics

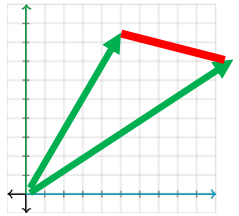
- 3 metrics commonly used to determine similarity of two vectors (2-dimensional representation)



Cosine similarity - measure the angle between two vectors; values from -1 to 1; 1 = both point in same direction; -1 point in opposite directions; 0 = orthogonal (perpendicular)



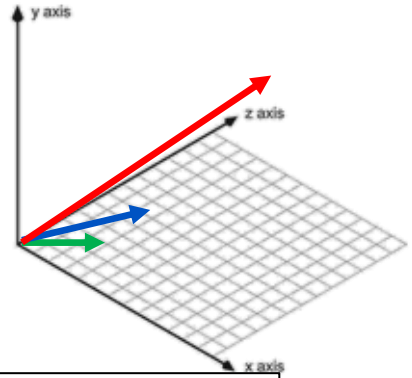
Dot product / inner product - measures how well 2 vectors align with each other; values from - ∞ to ∞; positive values indicate vectors are in same direction; negative values indicate opposite directions; 0 = orthogonal



Euclidean distance - measures the distance between two vectors; values from 0 to ∞; 0 = identical; larger numbers farther apart

imagine 3 vectors - a,b,c

$$a=\begin{bmatrix}.01\\.07\\.1\end{bmatrix} \quad b=\begin{bmatrix}.01\\.08\\.11\end{bmatrix} \quad c=\begin{bmatrix}.91\\.57\\.6\end{bmatrix}$$



Cosine similarity

$$sim(u,v)=\frac{u\cdot v}{\|u\|\|v\|}=\frac{\sum_{i=1}^{n}a_n b_n}{\sqrt{\sum_{i=1}^{n}u_n^2}\sqrt{\sum_{i=1}^{n}v_n^2}}$$

$$sim(a,b)=\frac{(a_1*b_1)+(a_2*b_2)+(a_3*b_3)}{\sqrt{a_1^2+a_2^2+a_3^2}\sqrt{b_1^2+b_2^2+b_3^2}}$$

$$=\frac{(0.01*0.01)+(0.07*0.08)+(0.1*0.11)}{\sqrt{0.01^2+0.07^2+0.1^2}\sqrt{0.01^2+0.08^2+0.11^2}}$$

0.0141

Dot product / inner product

$$u\cdot v=|u||v|cos\theta=\sum_{i=1}^{n}a_n b_n$$

Dot product formula

$$a\cdot b=(a_1b_1)+(a_2b_2)+(a_3b_3)$$

$$=(0.01*0.01)+(0.07*0.08)+(0.1*0.11)$$

0.0167

Euclidean distance

$$d(u,v)=\sqrt{\sum_{i=1}^{n}(u_i-v_i)^2}$$

Euclidean distance formula

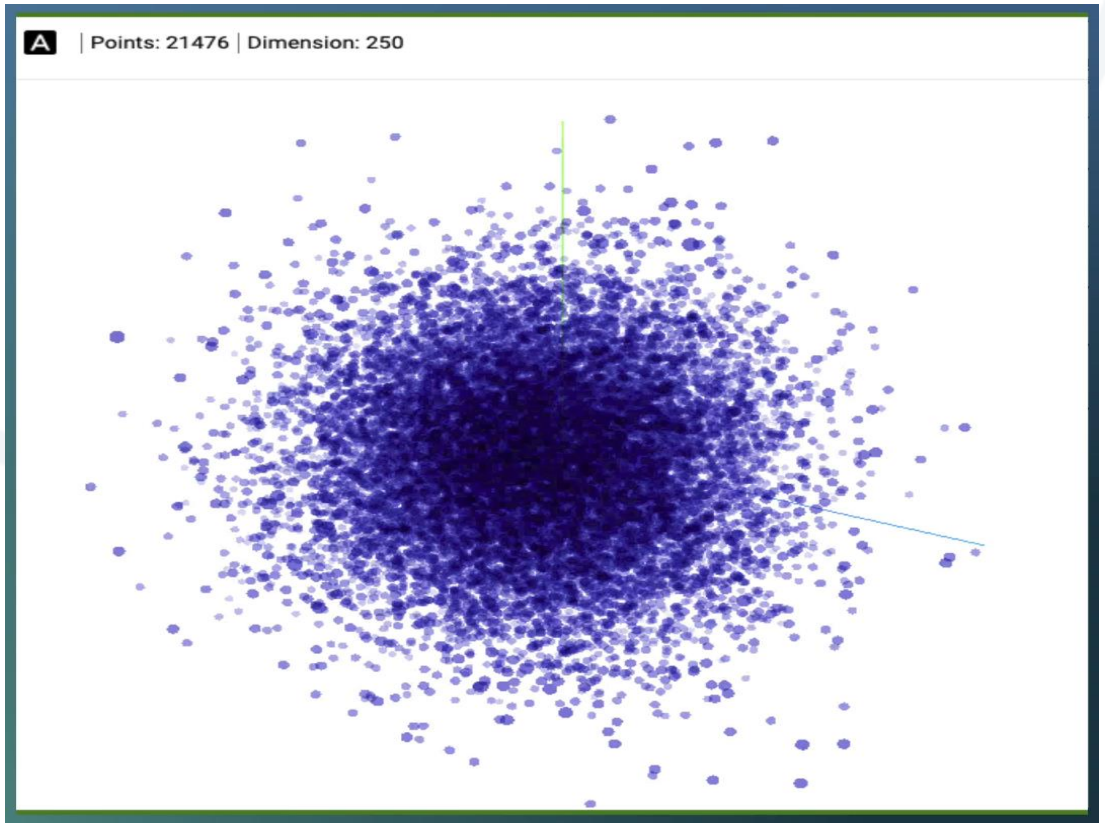
$$d(a,b)=\sqrt{(b_1-a_1)^2+(b_2-a_2)^2+(b_3-a_3)^2}$$

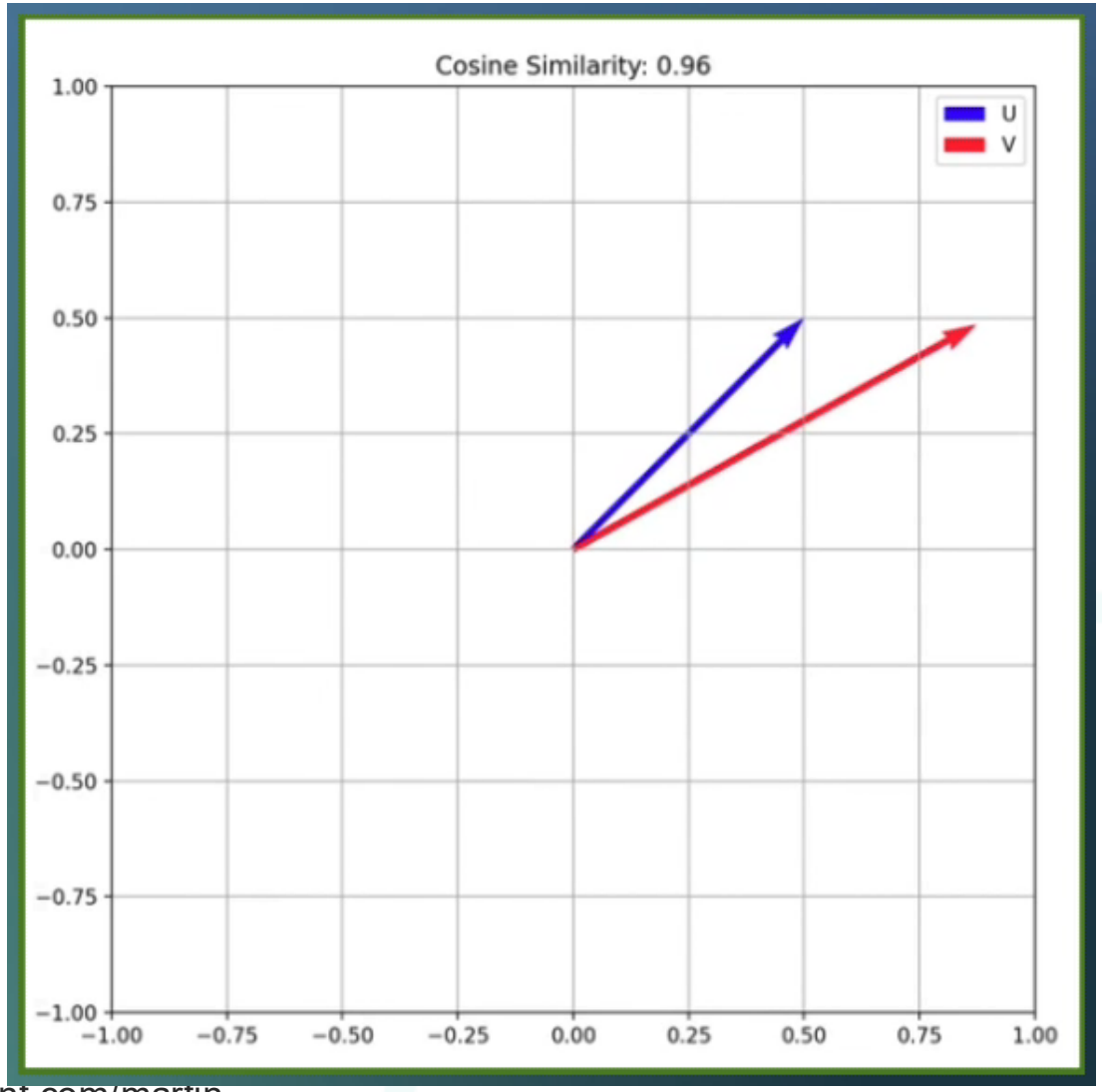
$$=\sqrt{(0.01-0.01)^2+(0.08-0.07)^2+(0.11-0.1)^2}$$

0.9998

Credit: https://towardsdatascience.com/similarity-metrics-in-nlp-acc0777e234c

Visualizing Embeddings and Vector Similarity





source: https://projector.tensorflow.org/?config=https://gist.githubusercontent.com/martin-labrecque/4483ff5a104f0b56417585c3bc9a12f1/raw/57348e12a70c8d70c2c573d3dbc0122ac077556b/journaux_config.json

Vectors and relationships example

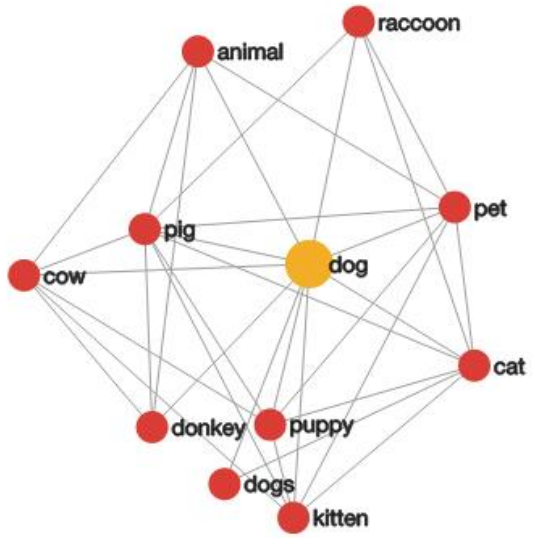
- Query - what words are related to "dog" in model "English Wikipedia"?

Word frequency

☑ High ☑ Medium ☐ Low

1. puppy NOUN 0.6909
2. cat NOUN 0.6893
3. pet NOUN 0.6779
4. pig NOUN 0.6680
5. dogs NOUN 0.6174
6. animal NOUN 0.6066
7. donkey NOUN 0.6060
8. raccoon NOUN 0.6021
9. cow NOUN 0.5953
10. kitten NOUN 0.5947

- *We show only the associates of the same part of speech as your query. All associates can be found at the Similar Words tab.*



Show the raw vector of «dog» in model MOD_enwiki_upos_skipgram_300_2_2021:

[-0.03301828354597092, 0.05134638026356697, 0.0036009703762829304, -0.04066073149442673, 0.10361430048942566, 0.013021323829889297, 0.028161464259028435, -0.0027567853685468435, 0.03388035297393799, -0.044882044196128845, 0.005169689189642668, -0.05818631127476692, 0.0533536821603775, 0.016616210341453552, 0.02030780538916588, -0.008570297621190548, -0.10925538837909698, -0.0708925873041153, 0.04675082117319107, -0.03091960959136486, -0.05172094330191612, 0.04471702128648758, 0.008674593642354012, -0.01816382259130478, 0.05909318849444389, 0.10409023612737656, 0.05633684620261192, -0.024881813675165176, 0.01872968301177025, 0.007228093687444925, -0.023127363994717598, 0.01528552919626236, -0.0643191784620285, -0.010359424166381359, -0.06104437634348869, -0.13868044316768646, -0.023004498332738876, 0.0038427673280239105, -0.021551262587308884, -0.03467748314142227, 0.010687021538615227, -0.017304275184869766, 0.026886526495218277, -0.0030398862436413765, -0.03685504570603371, -0.06017328053712845, 0.047442398965358734, -0.10714898258447647, 0.14808930456638336, -0.06579480320215225, -0.004342162515968084, 0.06226382404565811, 0.08031187951564789, -0.055930640548467636, -0.07030591368675232, 0.015474628657102585, 0.05367768555879593, 0.0917837843298912, 0.031899698078632355, 0.055091146379709244, -0.025078952312469482, -0.048126623034477234, -0.09730836749076843, -0.07128141075372696, 0.019415033981204033, -0.025872433558106422, -0.01761292852461338, 0.015608762390911579, -0.029876720160245895, -0.008602319285273552, 0.049825914204120636, 0.06784739345312119, 0.005586292129009962, -0.07148509472608566, -0.03097137063741684, -0.020296750590205193, 0.05099814385175705, 0.14920306205749512,

Source: http://vectors.nlpl.eu/explore/embeddings/en/MOD_enwiki_upos_skipgram_300_2_2021/dog_NOUN/

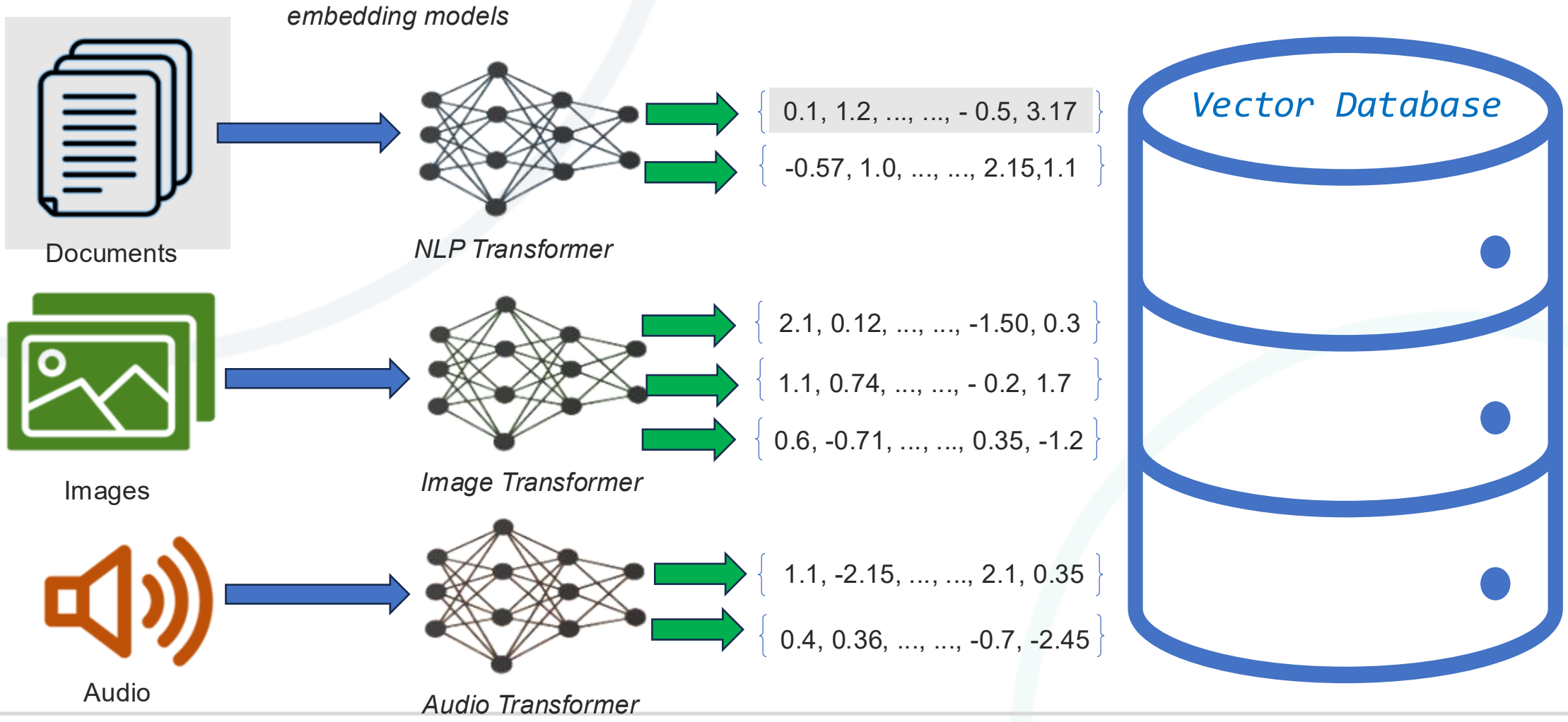
Vector Databases

Vector Databases

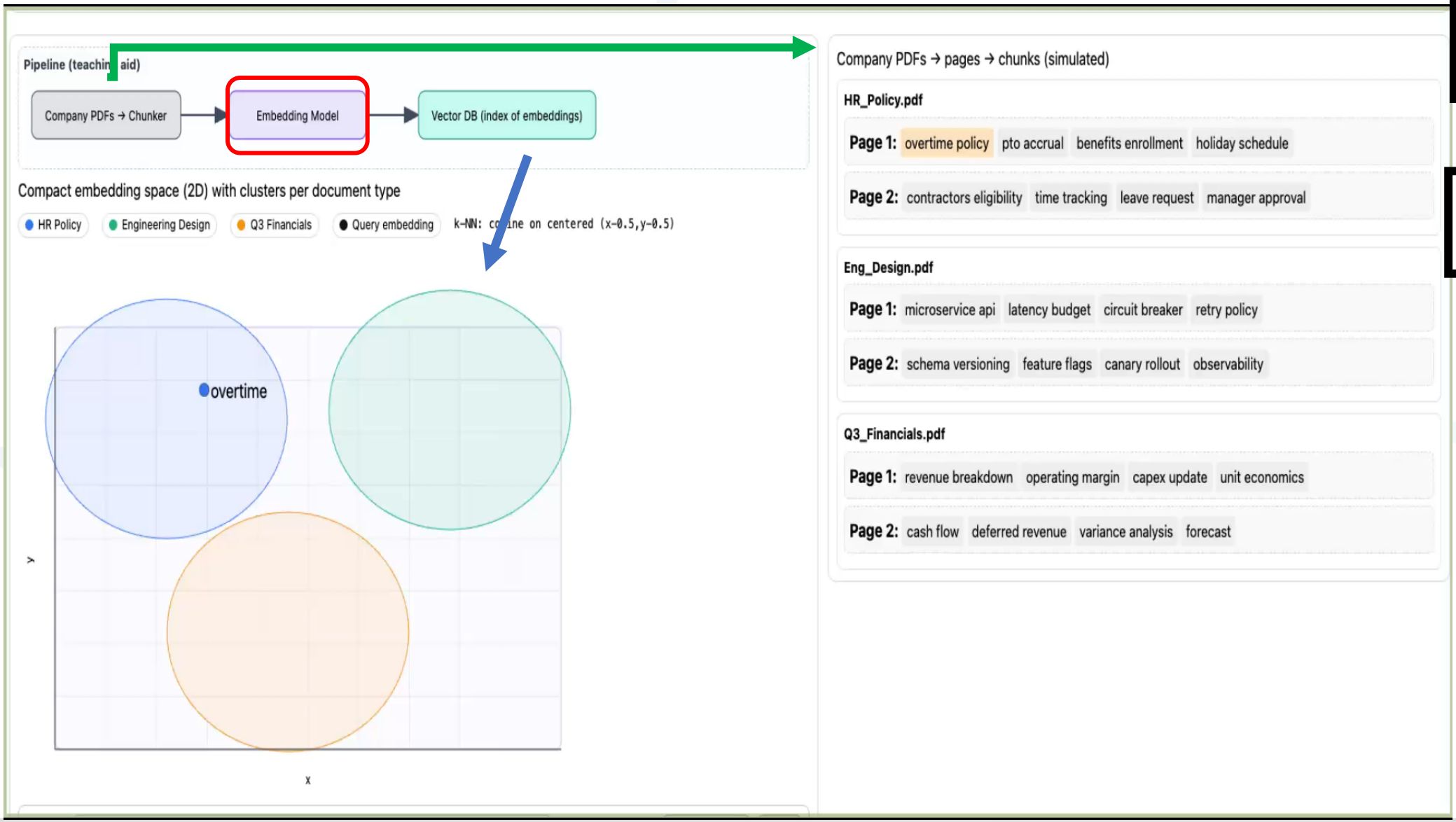
- Specialized database that index and stores *vector embeddings*
- Useful for
  - fast retrieval
  - similarity search
- Offer comprehensive data management capabilities
  - metadata storage
  - filtering
  - dynamic querying based on associate metadata
- Scalable and can handle large volumes of vector data
- Support real-time updates
- Play key role in AI and ML applications

How data gets into Vector Databases

- Data is input, converted to embeddings (vectors) and stored
- Queries are input, converted to embeddings (vectors) and then **similarity metrics** are used to find results ("nearest neighbors")



RAG prep – parsing your data and storing as relationships



Our Prompts also becomes Vectors

- Input: 'How do I reset my password?'
- ↓
- Model: Sentence Transformer (all-MiniLM-L6-v2)
- ↓
- Output: [0.12, -0.34, 0.56, ...] (384-dim vector)

- The Model:
  - Fast and efficient
  - Trained on millions of sentence pairs

Lab 2 – Working with Vector Databases

Purpose: *In this lab, we'll learn how to use vector databases for storing data and doing similarity searches*

Enriching with Metadata

Every chunk stores context about where it came from.

```
metadata = { "source": "Returns_Policy.pdf",
"page": 5, "type": "text", # or "table"
"chunk_index": 12 }
```

Why Metadata Matters?

- **Citations:** Tell the user exactly which page the answer came from.
- **Filtering:** "Search only in 'text' chunks" or "Search only this file".
- **Debugging:** Trace bad answers back to source.

Indexing: Metadata

PDF

- Each chunk stores:
- - Source filename
- - Page number
- - Content type (text vs table)
- - Chunk index

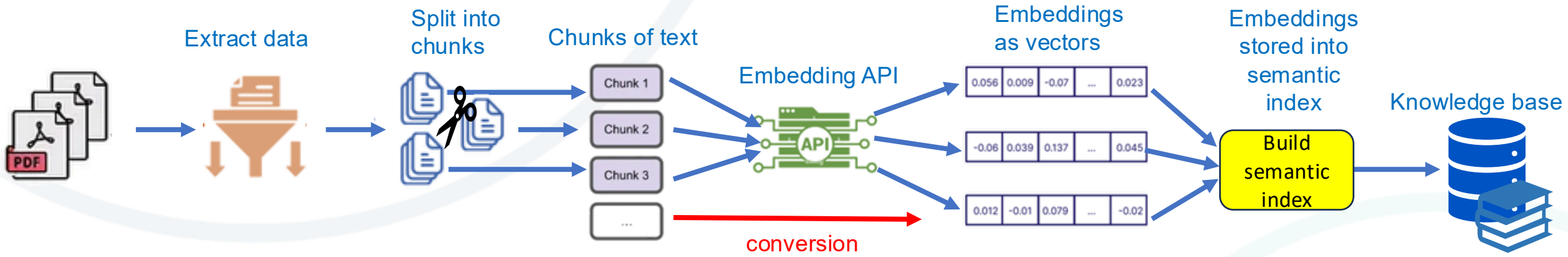
- Usage:
- - Source attribution
- - Citations
- - Filtering

Code

- Metadata stored:
  - file_path
  - language
  - start_line
  - end_line

- Enables precise linking:
  - Found in: src/auth/login.py:42-68

Detailed Data Ingestion Pipeline (Vector DB)



Stores embeddings / chunks

ChromaDb, Pinecone, Azure Index, Neo4j, etc. AKA "vector store"

Common PDF Parsing Frameworks

Tool / Framework	What it does (for RAG over PDFs)
Unstructured.io	Open-source, layout-aware PDF -> clean text/elements
LlamaParse	Hosted, high-fidelity PDF -> Markdown/text for complex layouts
Docling	OSS, high-quality parsing for complex, layout-heavy PDFs
Marker	PDF -> Markdown with OCR, good for long-form text
PageMage	Research-grade parser for scientific PDFs with structure
pypdf / PyPDF2	Simple Python PDF text extraction for basic docs
pdfplumber	Text + coordinates and table extraction
LangChain	RAG framework with multiple PDF loaders and chunkers
LlamaIndex	RAG framework with nod—based PDF ingest and chunking
Nanonets/Firecrawl	Managed APIs that output RAG-ready chunks from PDFs

UNSTRUCTURED.IO

LangChain

Docling

LlamaIndex

PyPDF2

Key Takeaways

- 1. Vectors represent meaning (Semantic Search)
- 2. Vector Databases allow fast similarity lookup
- 3. Chunking strategy is critical (Text vs Code)
- 4. Metadata allows for precise citation
- 5. Quality Retrieval = Quality Generation (RAG)

Using prompt templates to incorporate retrieved chunks

- A scaffold or blueprint that guides the AI in generating a response. It usually contains placeholders or variables that can be dynamically filled based on specific input data
- Optimizing prompt templates is a crucial technique for improving the performance of AI models
- A well-designed prompt template not only provides clear instructions to the model but also structures the interaction in a way that the model can best utilize its training
- Prompt templates can direct the model in how to factor in (or not) retrieved data

Only use retrieved data

```
PROMPT_TEMPLATE = """
Answer the question based only on the following context:
{context}
Answer the question based on the above context: {question}.
Provide a detailed answer.
Don't justify your answers.
Don't give information not mentioned in the CONTEXT INFORMATION.
Do not say "according to the context" or "mentioned in the context" or similar.
"""
```

Augment LLMs query with retrieved data

```
PROMPT_TEMPLATE = """
Answer the question: {question} using whatever resources you have.
Include any related information from {context} as part of your answer.
Provide a detailed answer.
Don't justify your answers.
"""
```

Putting It All Together: The RAG Flow

1. INDEX PHASE (One time)

    - Extract -> Chunk -> Embed -> Store

2. SEARCH PHASE (Every query)

    - Query -> Embed -> Retrieve

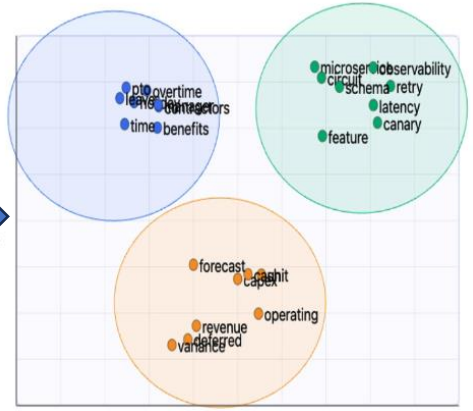
3. GENERATION PHASE (RAG)

    - Retrieved Chunks + Prompt -> LLM -> Answer

RAG – Retrieval Augmented Generation

Query: overtime policy for contractors

AI Response: Here's some information on our overtime policy for contractors and how overtime works…



- HR_Policy.pdf p.2 — "time tracking" cos≈1.00
- HR_Policy.pdf p.2 — "leave request" cos≈1.00
- HR_Policy.pdf p.1 — "benefits enrollment" cos≈0.99

Augmented Prompt

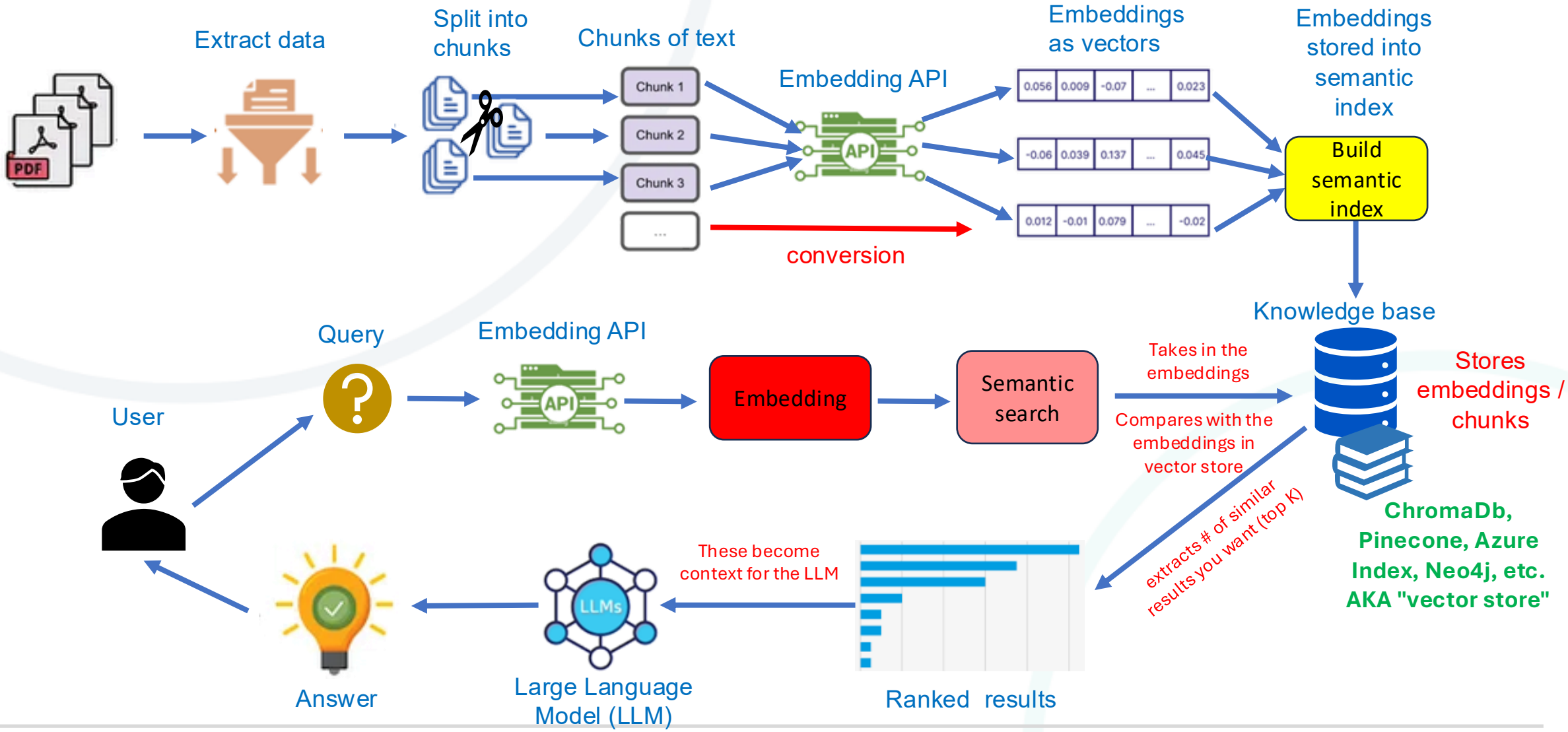
```
Question:
overtime policy for contractors

Context (retrieved):
- (HR_Policy.pdf p.2) time tracking
- (HR_Policy.pdf p.2) leave request
- (HR_Policy.pdf p.1) benefits enrollment

Instruction:
```

LLM

Lab 3 – Working with RAG implemented with vector dbs

Purpose: *In this lab, we'll build on the use of vector databases to parse a PDF and allow us to include it in context for LLM queries*

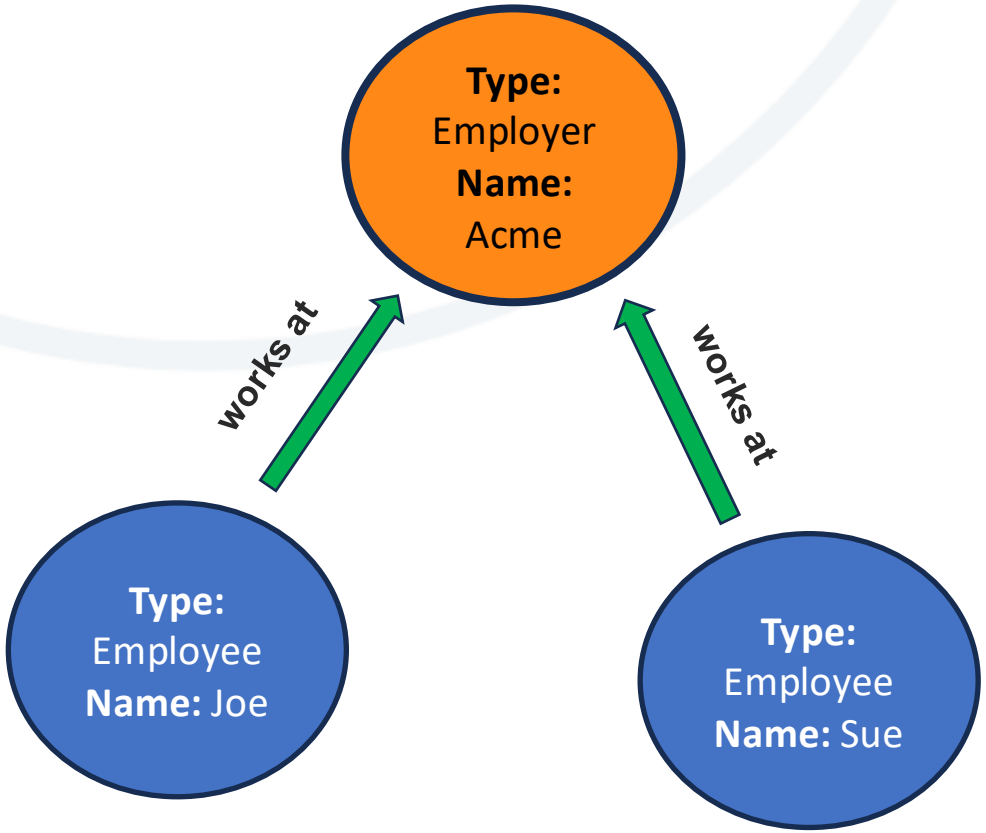
Typical RAG Pipeline (Vector DB)



Knowledge Graphs

What is a Knowledge Graph?

- Structure to represents info in relational, interconnected format
- entities = nodes, relationships = edges
- Can find new information by exploring relationships



Query #1: Where does Sue work?

Answer #1: Sue works at Acme

Question #2: Who works for Acme?

Answer #2: Joe and Sue work for Acme.

Question #3: Does Sue work for the same company as Joe?

Answer #3: Yes, Sue works for the same company as Joe.

Knowledge Graph Challenges

- Construction - need to identify and extract relationships, entities from data sources and integrate them into a graph
- Data consistency and compatibility - data can come from multiple sources; needs to be made consistent, have conflicts resolved, cleaning etc.
- Maintenance and evolution - graphs need to be continuously updated and maintained
- Performance and scalability - growth in size and complexity can be difficult to manage; optimization becomes critical
- Complexity of queries - crafting complex queries can be challenging
- No standardization - lack of compatibility and lock-in to app/vendor
- Explainability - for users
- Domain complexity - terminology, complex concepts, etc.

Graph RAG

Graph RAG in AI

- RAG that incorporates knowledge graphs or graph databases
- AI retrieves information not just from unstructured text docs but from structured graph data
- Allows model to leverage rich relationships and entities encoded within graphs

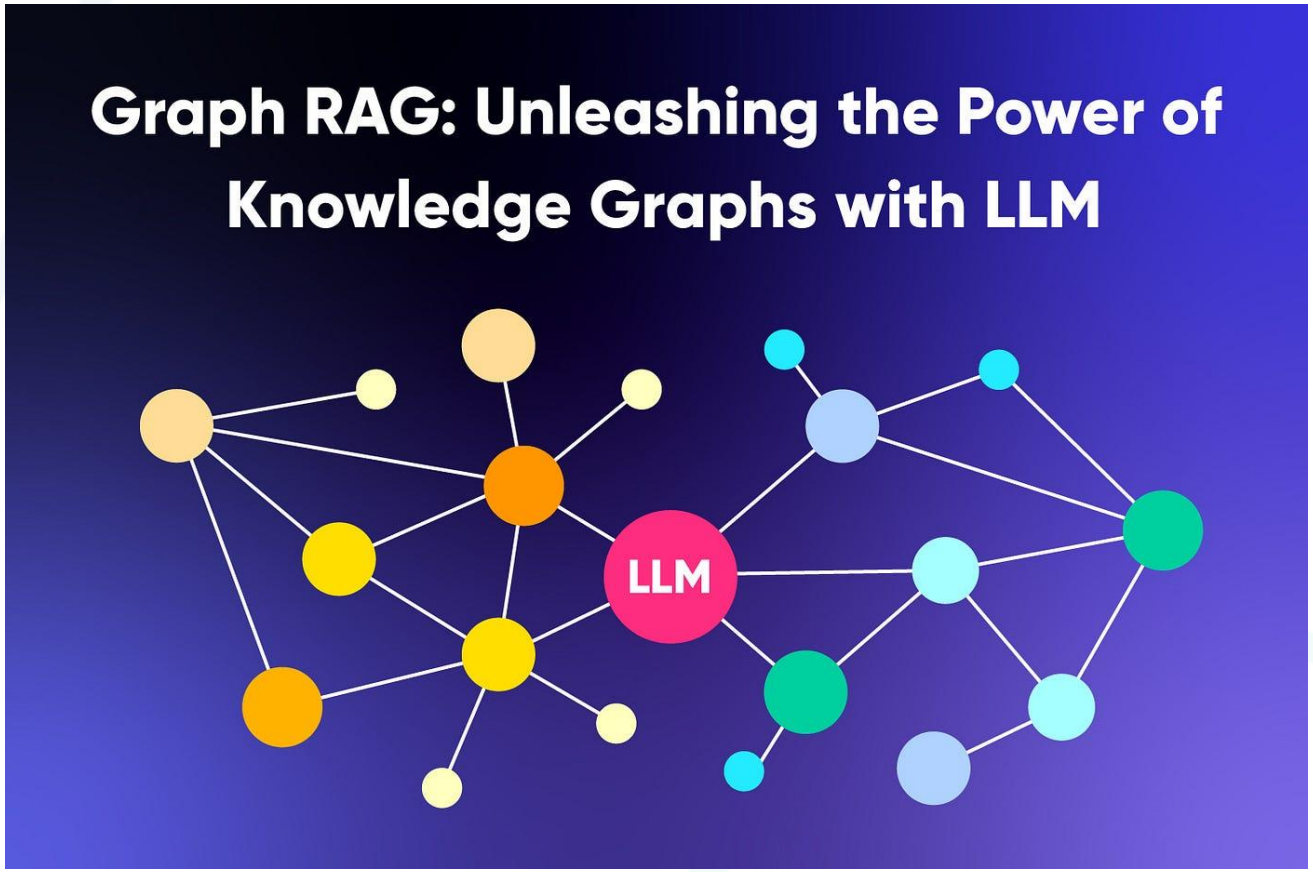
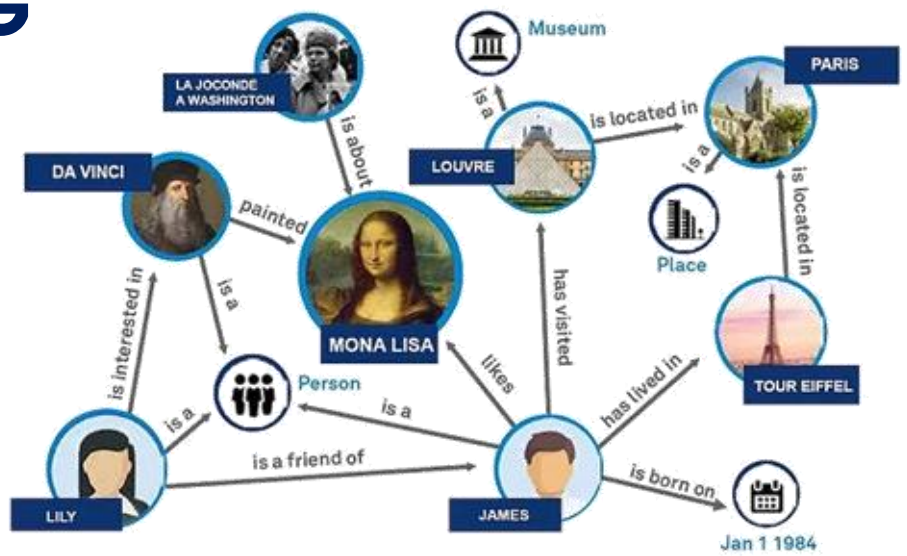


Image credit: https://medium.com/@nebulagraph/graph-rag-the-new-llm-stack-with-knowledge-graphs-e1e902c504ed

Key Components of Graph RAG

- **Graph-based knowledge**: Information is stored as entities (nodes) and their relationships (edges).
- **Graph retrieval**: Queries fetch relevant nodes or subgraphs related to the question.
- **Augmented generation**: The retrieved graph data is used by the language model to produce better answers.

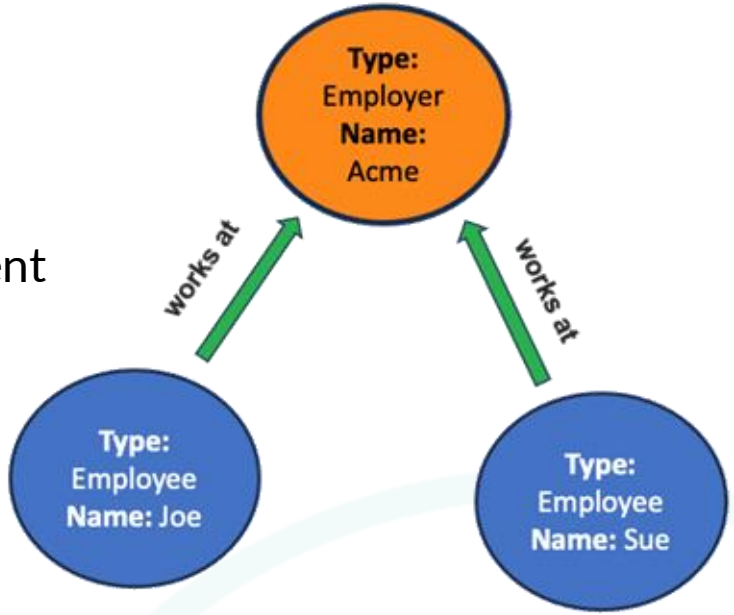



```
def query_graph():
    query = """
    MATCH (p:Person)-[:WORKED_WITH]->(p2:Person)
    RETURN p.name AS person, p.profession AS profession, p2.name AS collaborator
    """
    result = graph.run(query).data()
    return result
```

```
# Prepare a prompt from the graph result
prompt = "Generate a biography based on the following information:\n\n"
for record in result:
    prompt += f"{record['person']} is a {record['profession']} who worked with {record['collaborator']}.\n"
```

Benefits of using Knowledge Graphs in RAG

- **Structured knowledge**
  - Information is stored as entities and relationships (nodes and edges).
  - *Example:* A "works at" relationship shows who works for Acme.
- **Better context**
  - Relationships help the system understand how things are connected.
  - *Example:* Knowing Sue and Joe both "work at" Acme helps answer employment questions.
- **Inferential reasoning**
  - Following relationships allows the system to infer new facts.
  - *Example:* The system can infer Sue and Joe work for the same company.
- **Knowledge integration**
  - Graphs can combine data from multiple sources into one view.
  - *Example:* Adding data about other companies and employees enriches answers.
- **Transparency and explainability**
  - The graph makes it easier to explain how an answer was derived.
  - *Example:* "Sue and Joe both have a 'works at' relationship with Acme."



Applications of Graph RAG

- **Question Answering Systems**: Providing precise answers by querying knowledge graphs for specific facts or relationships.
- **Dialogue Systems**: Enhancing conversational agents with context-aware responses based on user history or preferences stored in graphs.
- **Recommendation Systems**: Generating personalized recommendations by traversing user-item interaction graphs.

Example Workflow for Graph RAG

User Input: A user asks, "What movies directed by Christopher Nolan feature time travel?"

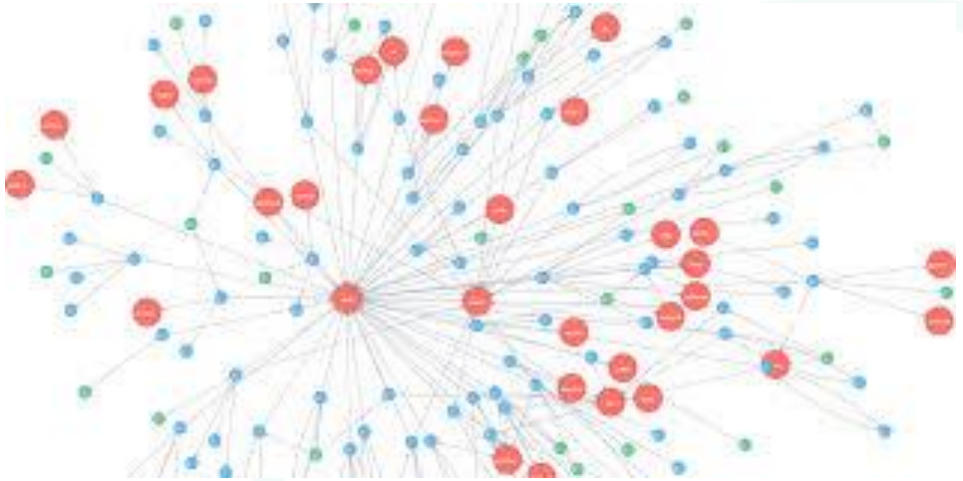
Graph Retrieval: The system queries a movie knowledge graph to find nodes connected to "Christopher Nolan" with relationships indicating "directed" and retrieves movies associated with "time travel."

Generation: Using the retrieved information, the model generates a response: "Christopher Nolan directed 'Interstellar,' which features themes of time travel."

Neo4j

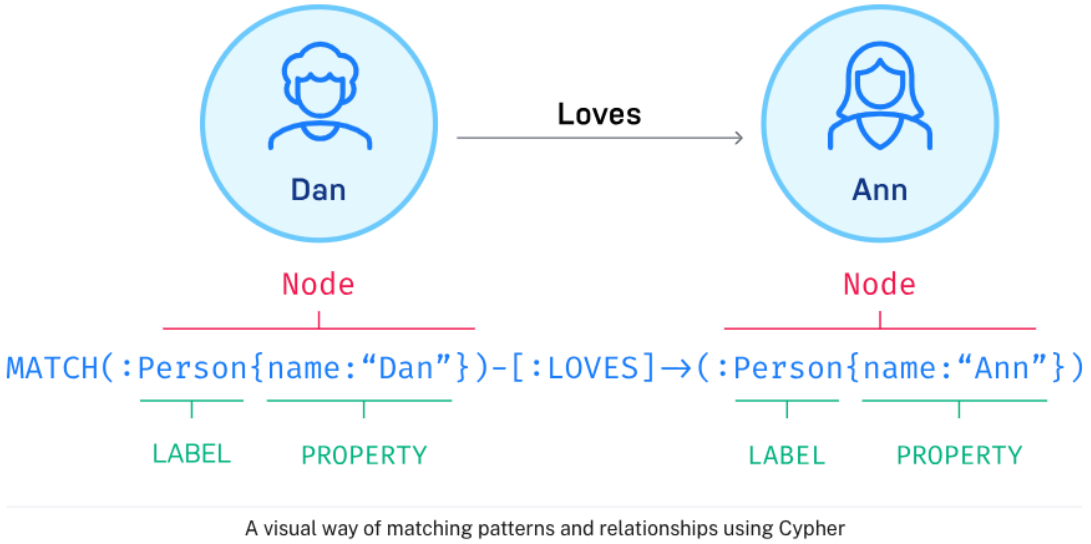
What is Neo4j?

- **Graph database**
  - Neo4j stores data as nodes (entities) and relationships.
- **Cypher query language**
  - Uses Cypher, a SQL-like language designed for graph queries and traversals.
- **Fast for connected data**
  - Optimized for exploring relationships, paths, and complex connections.
- **Scalable and reliable**
  - Supports clustering and scaling for large datasets and high availability.
- **Common use cases**
  - Fraud detection, recommendations, social networks, and knowledge graphs.



Cypher

- Neo4j's graph query language (like SQL for graphs)
- Provides a way of matching patterns and relationships
  - graph model is made of nodes and relationships
  - nodes and relationships together make up "patterns"
  - Cypher is designed to recognize versions of patterns in data
  - can do simple or complex path traversals
- Has its own open source implementation (openCypher)
- Basic of graph query language (GQL) standard



A visual way of matching patterns and relationships using Cypher

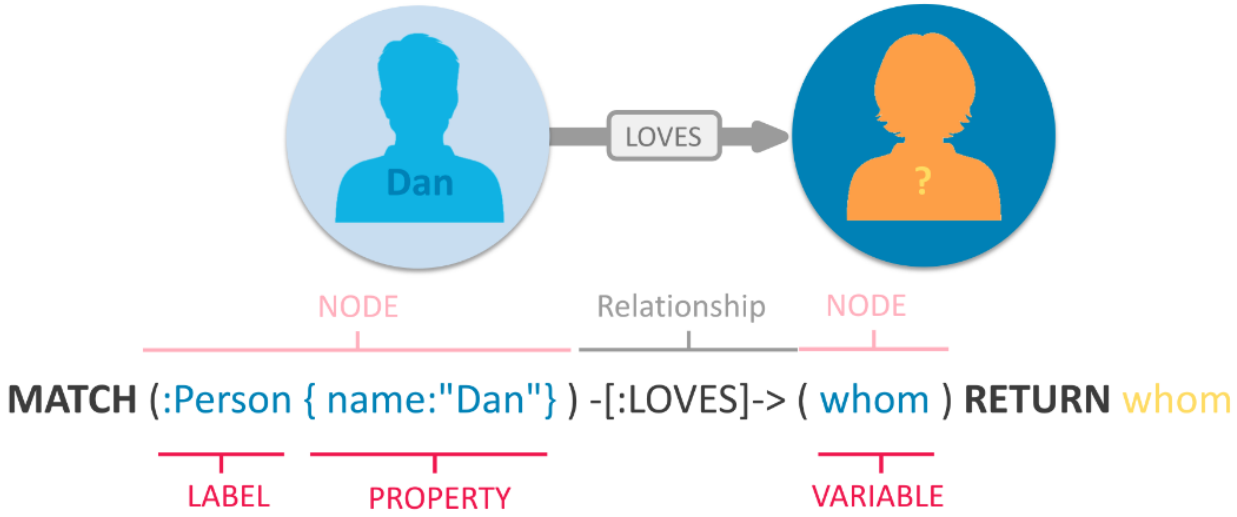


Image credit: https://neo4j.com/product/cypher-graph-query-language/

Cypher vs SQL

- Similarities
  - Example - both use WHERE & ORDER BY
- More schema flexibility in Cypher
  - Cypher doesn't require a fixed schema
  - New attributes and relationships can be added as graphs evolve
- Cypher has different order for query
  - Returns values after finding relationships vs selecting values
- Cypher queries are more concise

```
SELECT movie.name
FROM movie
WHERE movie.rating > 7
```

```
MATCH (movie:Movie)
WHERE movie.rating > 7
RETURN movie.title
```

```
SELECT actors.name
FROM actors
        LEFT JOIN acted_in ON acted_in.actor_id = actors.id
        LEFT JOIN movies ON movies.id = acted_in.movie_id
WHERE movies.title = "The Matrix"
```

```
MATCH (actor:Actor)-[:ACTED_IN]->(movie:Movie {title: 'The Matrix'})
RETURN actor.name
```

Lab 4 – Implementing Graph RAG with Neo4j

Purpose: *In this lab, we'll* **we'll** *see how to implement Graph RAG by querying a Neo4j database and using Ollama to generate responses*

Frameworks for use with RAG

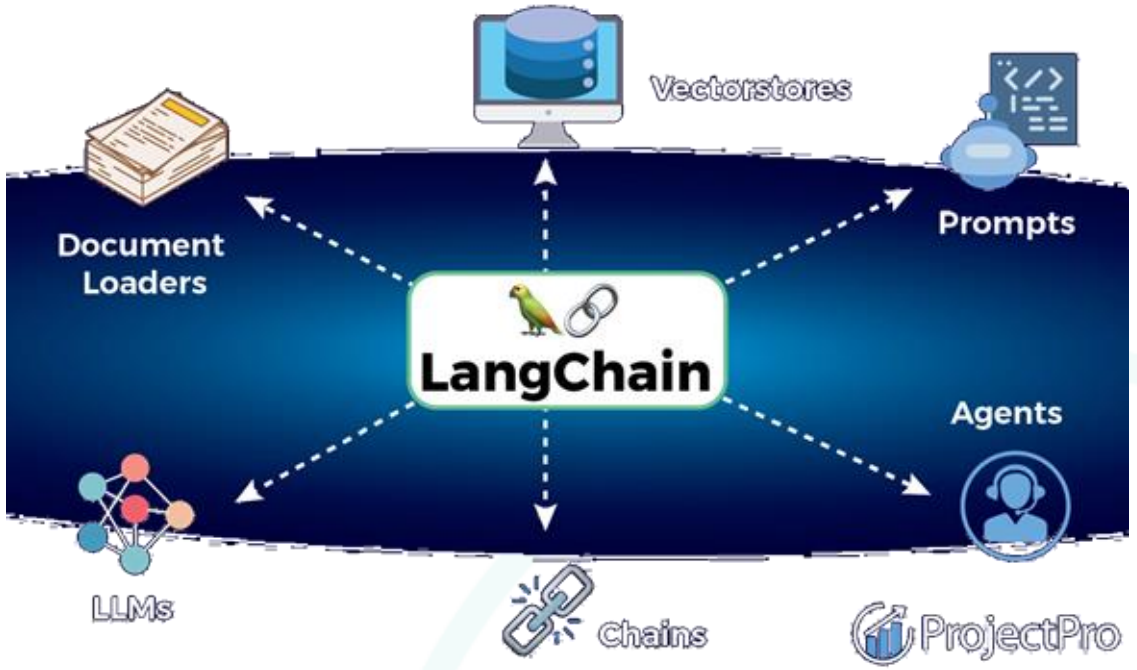
What are frameworks and why do we use them?

What they are

- Tools that help build AI apps using LLMs plus external data (documents, databases, APIs).

Why use them

- **Easy integration**
  - Connect LLMs to data sources with minimal setup.
- **Flexible pipelines**
  - Build modular retrieval and generation workflows.
- **Efficient retrieval**
  - Use built-in vector search and indexing for better results.
- **Smarter queries**
  - Combine retrieval and reasoning for more accurate answers.
- **Faster development**
  - Reuse pre-built components instead of building everything from scratch.



Framework selection

- **Data support**
  - Handles your document, structured, or graph data
- **Retrieval options**
  - Vector, graph, keyword, or hybrid search
- **Model flexibility**
  - Supports cloud and local LLMs
- **Ingestion & storage**
  - Indexing, metadata, and database integration
- **Quality & control**
  - Evaluation, observability, and security
- **Scalability & usability**
  - Performance, reliability, and developer experience

LlamaIndex

Haystack by deepset

LangChain

- **Summary**
  - Ideal for complex, multi-step pipelines with diverse integrations
- **Advantages**:
  - Modular (chain together) with broad integrations
  - Broader ecosystem integrations with APIs, LLMs, other external sources
  - Pre-built workflows
  - Support for tools
- **Disadvantages**:
  - More complex to learn and setup
  - Heavier for basic tasks; more overhead for simple tasks
- **Use Cases**:
  - Complex RAG applications with multi-step operations and diverse data sources
  - API/data integrations
  - Advanced conversational agents and knowledge retrieval systems

Overview

LangChain is a comprehensive framework designed to facilitate the development of applications that integrate LLMs with other data sources or external tools. It enables chaining different steps together to build complex pipelines for LLMs, including retrieval tasks, multi-step conversations, and more.

Fit for RAG

LangChain shines in more complex RAG scenarios where you need to combine document retrieval with multiple other steps or when pulling in data from diverse sources (e.g., APIs, tools).

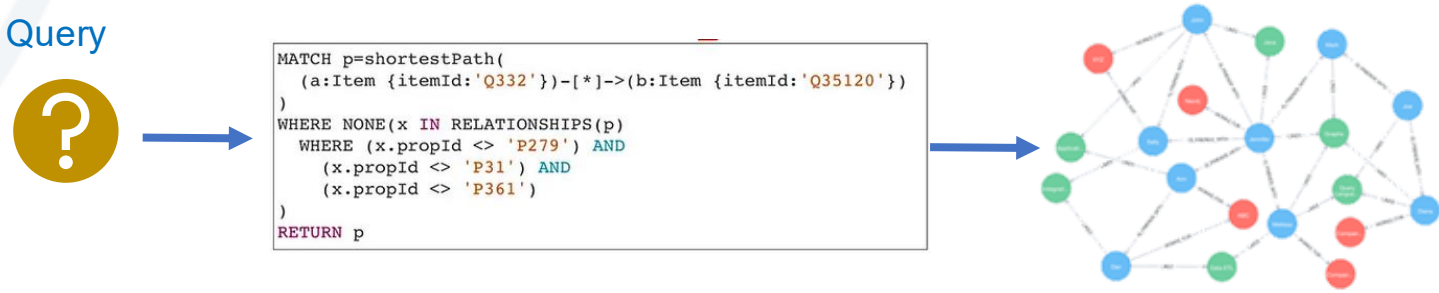
GraphCypherQAChain

What it is

- A LangChain component that connects natural language to graph databases (like Neo4j).

How it works

- Ask a question in plain language
- An LLM converts it into a Cypher query
- The query runs on the graph database
- Results are returned as a readable answer



Why it's useful

- No need to learn Cypher
- Easier access to graph data
- Faster application development

Common use cases

- Interactive data exploration
- Chatbots and virtual assistants
- Educational and learning tools

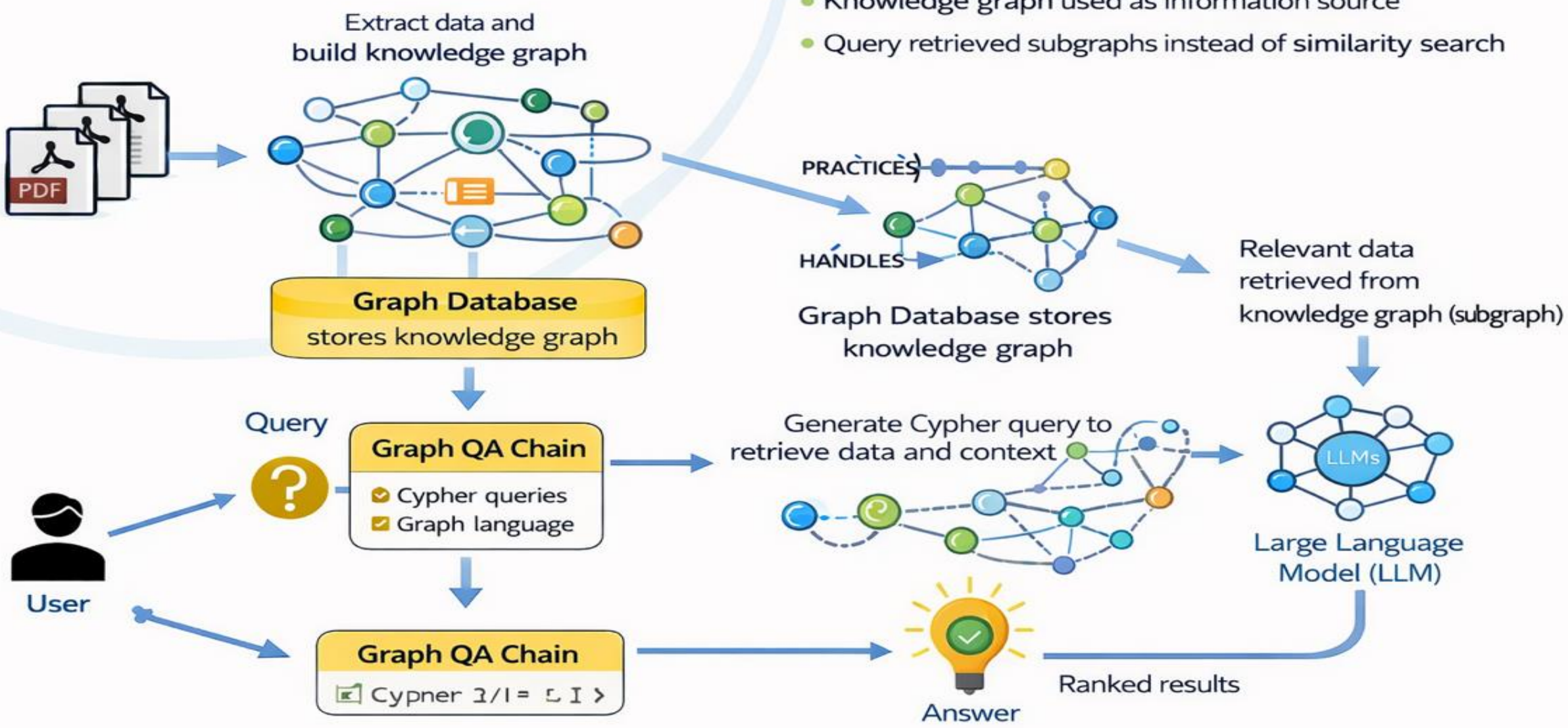
```
chain = GraphCypherQAChain.from_llm(
    cypher_llm=Ollama(model="llama3",temperature=0),
    qa_llm=Ollama(model="llama3",temperature=0),
    graph=graph, verbose=True,
)

response = chain.invoke({"query": "Who starred in Star Trek: Generations?"})
print(response["result"])
```

Lab 5 – Simplifying Graph RAG with Frameworks

Purpose: *In this lab, we'll see how to simplify Graph RAG by leveraging frameworks and using LLMs to generate queries*

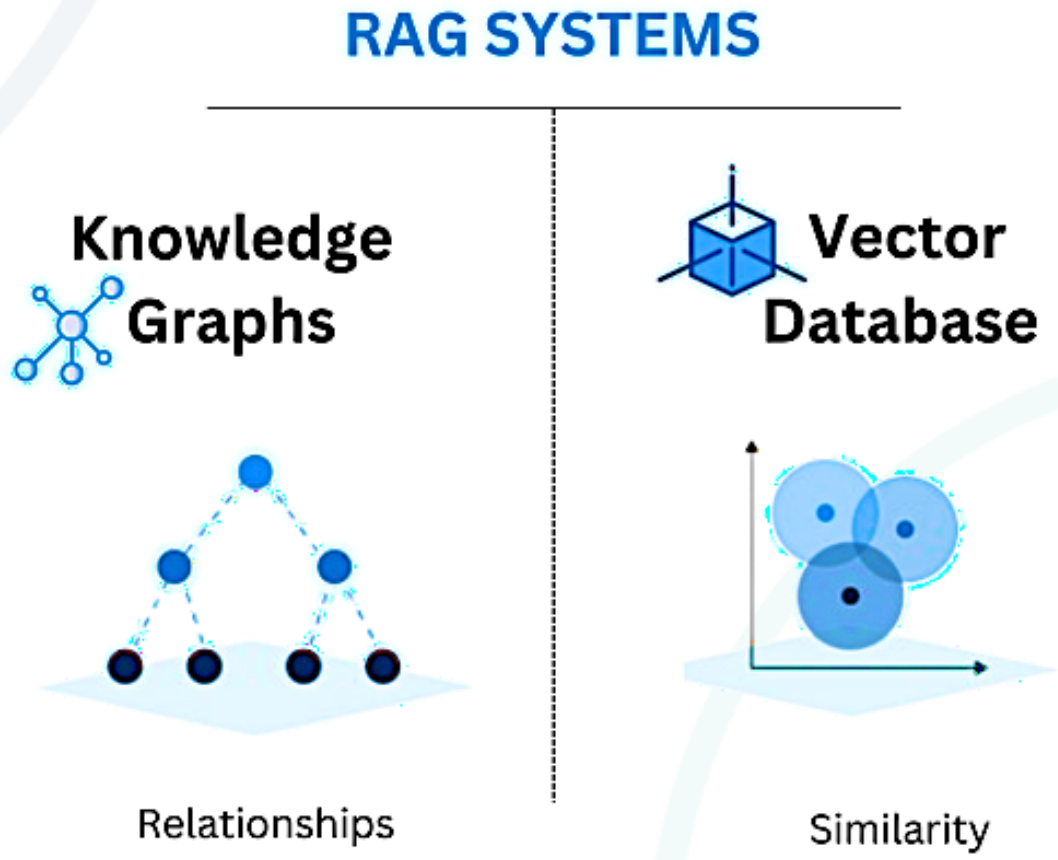
Typical Graph RAG Pipeline (Graph DB)

- Knowledge graph used as information source
- Query retrieved subgraphs instead of similarity search



Implementing RAG – Data Stores

- Data stores (databases) for RAG are primarily either:
  - Vector databases
  - Graph databases
  - Hybrid



Hybrid RAG

Semantic + Knowledge Graph

Combining Meaning and Structure for Complete Retrieval

Meaning vs Structure

Semantic Search

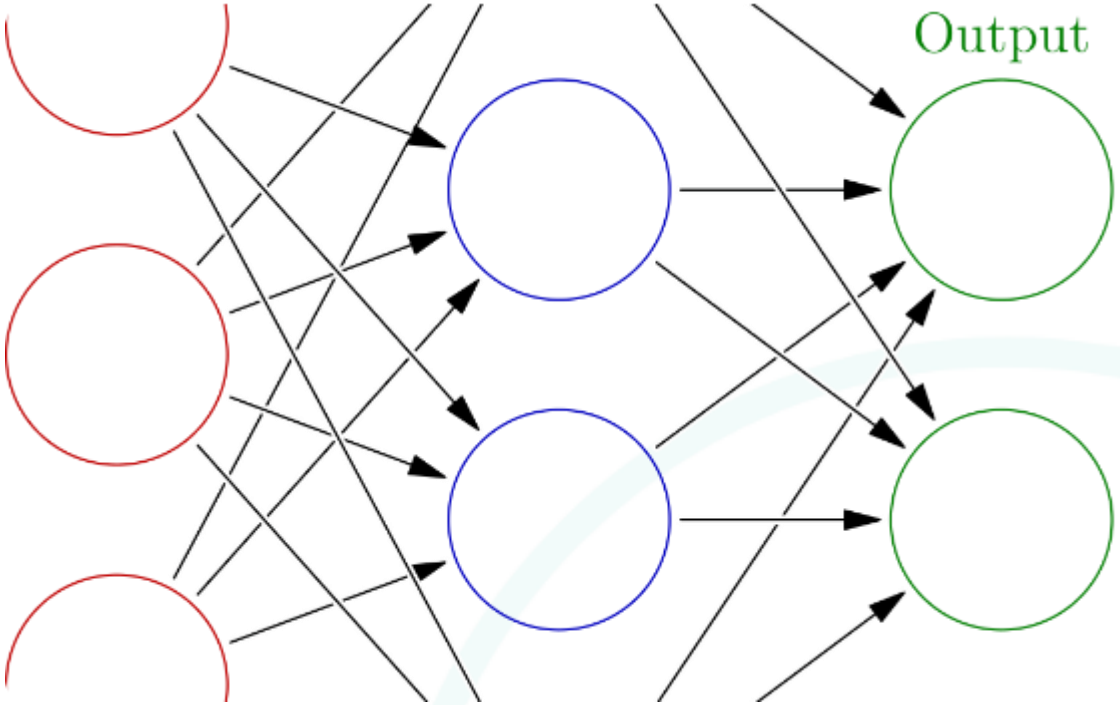
Understands MEANING: Handles natural language flexibility and vocabulary mismatches (e.g., "money back" → "refund").

Uses ChromaDB with vector embeddings to find conceptually related content.

Graph Search

Understands STRUCTURE: Maps explicit relationships between entities (e.g., Pro_Series → HAS_TIMEFRAME → 14_Days).

Uses Neo4j with Cypher queries for precision facts.



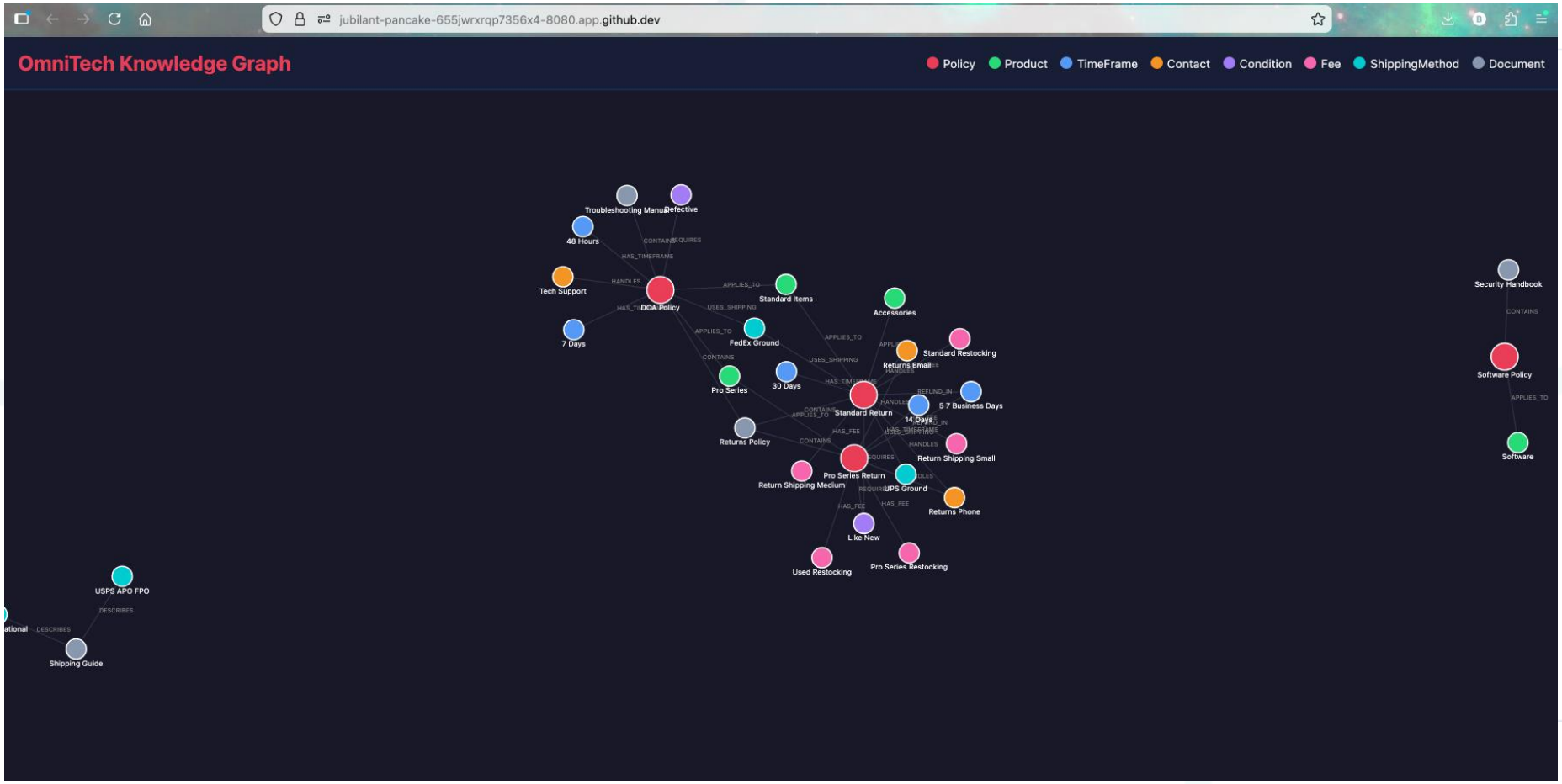
The OmniTech Knowledge Graph

A pre-built graph in Neo4j extracted from OmniTech PDFs:

36 Nodes & 48 Relationships

Key Entities: Products, Policies, Contacts, TimeFrames, Conditions.

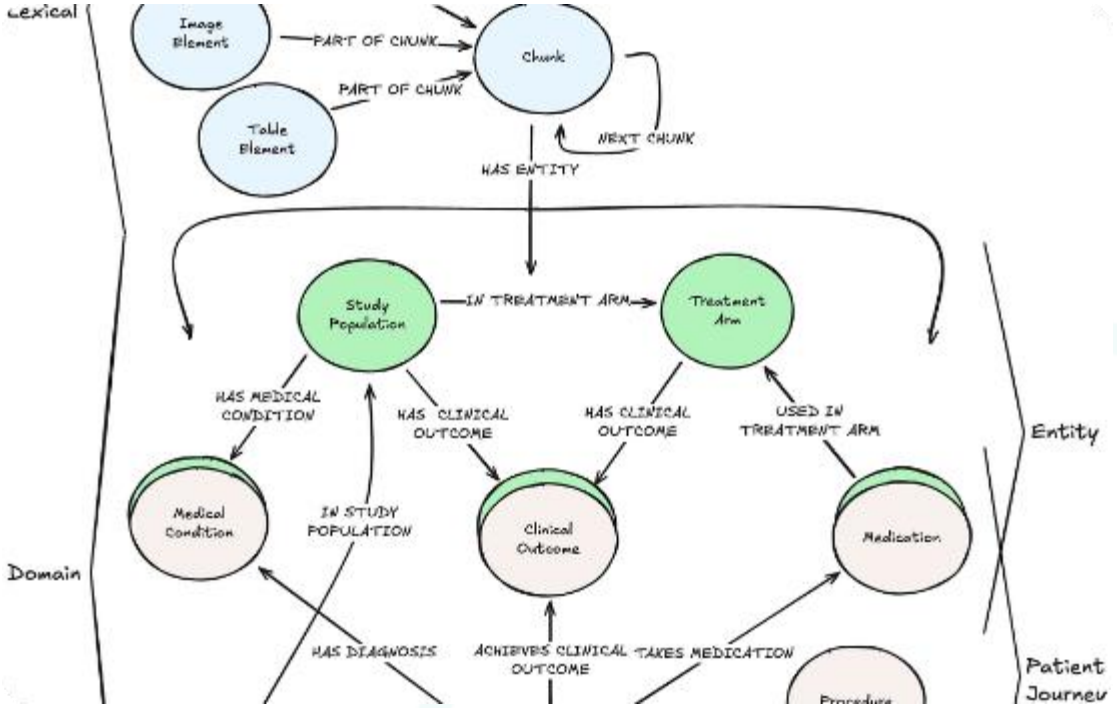
Relationships: APPLIES_TO, HAS_TIMEFRAME, HANDLES.



This allows the agent to traverse facts like a map rather than just searching for text chunks.

From Documents to Knowledge Graph

Approach	How it Works
Manual	Human identifies entities and relationships, structures into CSV/Cypher.
NER	NLP models identify entities (names, dates, orgs) automatically.
LLM	Prompting an LLM to "Extract entities and relationships" from text.



Same Query, Different Results

Query: "What's the return window for Pro-Series equipment?"

SEMANTIC (ChromaDB)

Finds text chunks where "return window" and "Pro-Series" appear.

```
"...Pro-Series designation have a strict 14-day return window..."
```

Requires LLM to parse and extract the answer.

GRAPH (Neo4j Cypher)

Traverses explicit edges to return the specific node value.

```
MATCH (p:Product {name: 'Pro_Series'})-[:HAS_TIMEFRAME]->(tf)
RETURN tf.value
```

Returns the direct fact: 14 days.

Traversing Relationships

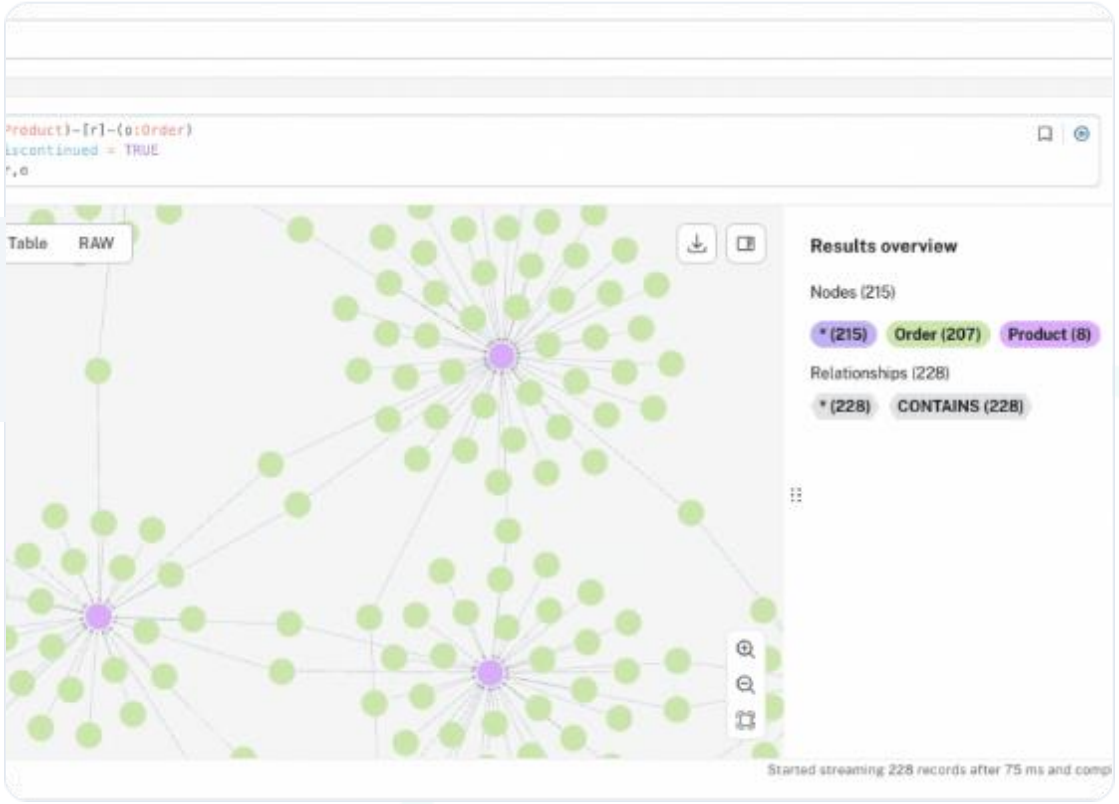
Graphs shine when connecting distant facts. Consider the query:

"Who do I contact about a defective item?"

```
MATCH (pol:Policy {name: 'DOA_Policy'})
OPTIONAL MATCH (c:Contact)-[:HANDLES]->(pol)
RETURN c.value, c.department
```

The graph followed the path:

DOA_Policy ← HANDLES ← Tech_Support

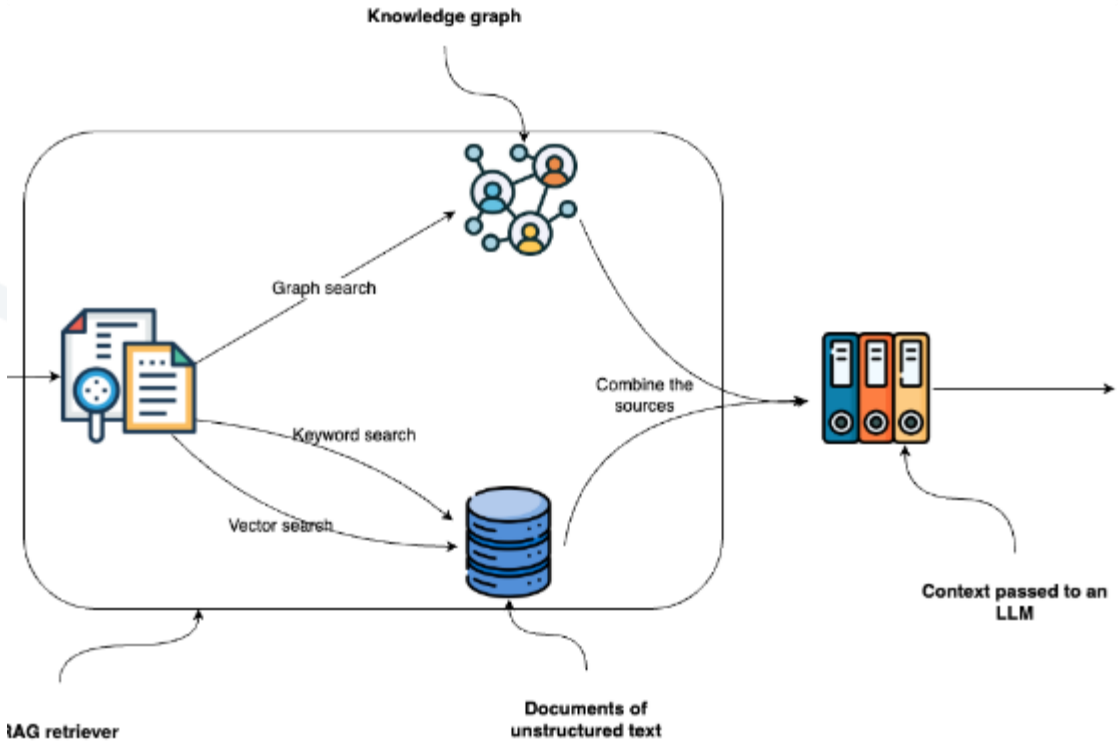


Why Combine Both?

Approach	Tool	Strength	Weakness
Semantic	ChromaDB	Natural language, context, synonyms	May miss precise structural data
Graph	Neo4j	Precise facts, relationship traversal	Only as good as the modeled data
Hybrid	**Both**	**Complete, precise, contextual answers**	Slightly more complex architecture

Hybrid Scenario: Use Graph for the 14-day window fact, and Semantic for the paragraph explaining exceptions.

The Hybrid RAG Architecture



1. **Query:** The question is sent to both databases.
2. **Retrieval:** Graph returns precise facts; Vector DB returns contextual chunks.
3. **Augment:** The LLM prompt is built using the union of both datasets.
4. **Generate:** Ollama synthesizes the final coherent answer.

Hybrid RAG: Key Takeaways

- **ChromaDB understands MEANING.** It handles natural language flexibility and synonyms.
- **Neo4j understands STRUCTURE.** It handles precise facts and explicit connections.
- **Together they provide COMPLETE answers.** The LLM synthesizes facts and context for high-quality generation.
- **Strategy:** Use Graph for facts/contacts and Semantic for explanations/exceptions.

Precision + Context = Production Grade RAG

Lab 6 Code Structure

- **Initialize Connections:** Set up both ChromaDB PersistentClient and Neo4j Graph connection in the HybridRAG class.
- **Semantic Search:** Execute collection.query() to retrieve context from vector embeddings.
- **Graph Search:** Execute graph.run() with Cypher to retrieve structured relationship data.
- **Hybrid Integration:** Merge graph_results and semantic_results to augment the LLM prompt.

Lab 6 – Hybrid RAG: Semantic Search + Knowledge Graph

Purpose: *In this lab,* ***we'll build a hybrid RAG system that combines semantic search (ChromaDB) with knowledge graph traversal (Neo4j) to get both precision and context in our answers.***

Hybrid RAG Pipeline (Graph + Vector DB)

Extract entities and document chunks, then build knowledge graph and embeddings

- Structured relationships
- Optional: attach embeddings to nodes for hybrid search

Graph Database
stores knowledge graph

Attach optional embeddings to graph nodes

- Search both by similarity and graph-based relationships

Extract entities and document chunks, then build knowledge graph and embeddings

API

Graph Search
- Cypher code
- Focused subgraphs

Vector Search
- Similarity search
- Top-k chunks

Query

Search both by similarity and graph nodes

User

Vector Database

Graph Database
stores knowledge graph

Vector Database

Vector Database
stores embeddings for semantic retrieval

LLMs

Answer

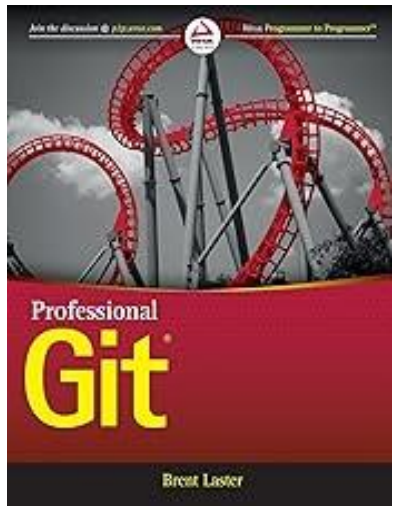
- **Hybrid Retrieval:** Retrieved subgraphs capture structured knowledge; top-k similar documents provide semantic context

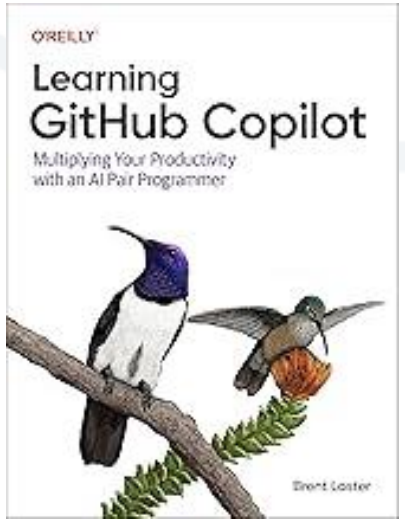
Choosing Your RAG Strategy

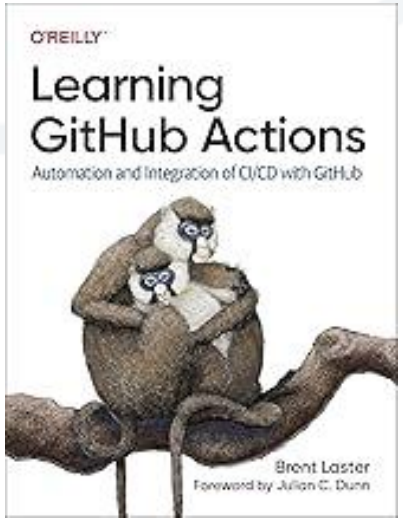
Use Case	Best Approach
Natural language Q&A	Semantic (ChromaDB)
Precise facts (dates, contacts)	Graph (Neo4j)
Relationship questions ("Who handles X?")	Graph (Neo4j)
Vocabulary mismatch ("money back" → refund)	Semantic (ChromaDB)

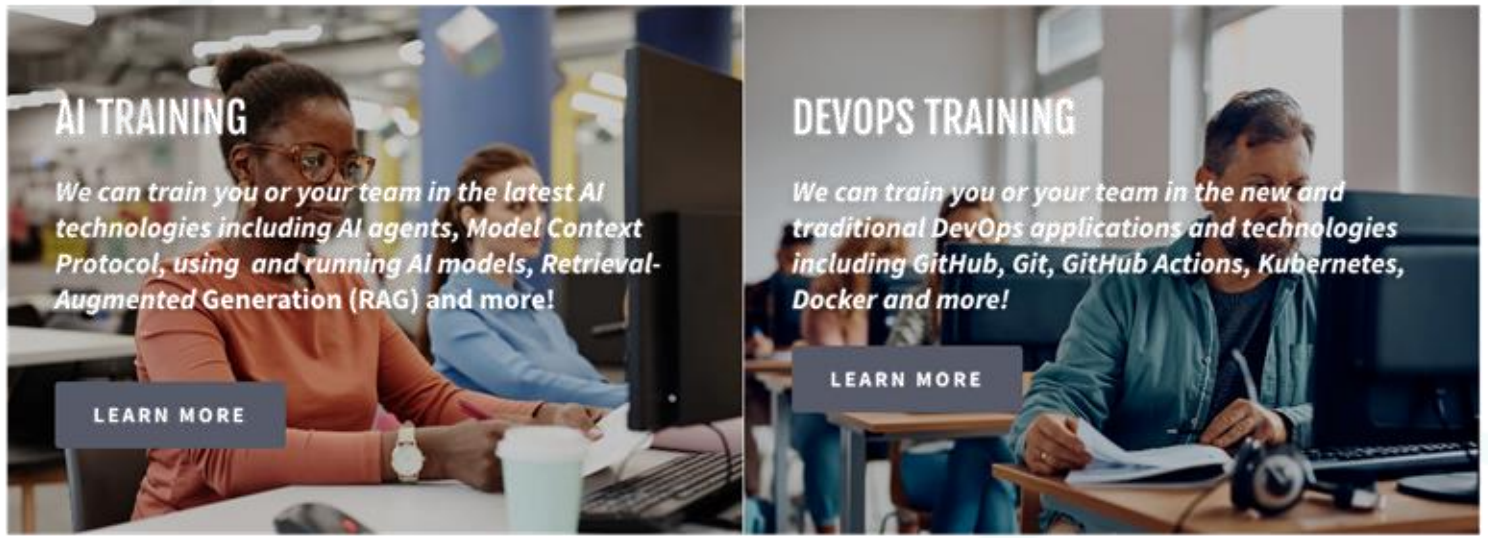
That’s all - thanks!

Contact: training@getskillsnow.com









techskillstransformations.com
getskillsnow.com

Tip: Persisting your codespace

- When signed in, go to github.com/codespaces
- Find codespace at bottom
- Click on "..."
- Select "Export changes to a branch"
- Will make a fork of original repo and put your changes into a new branch