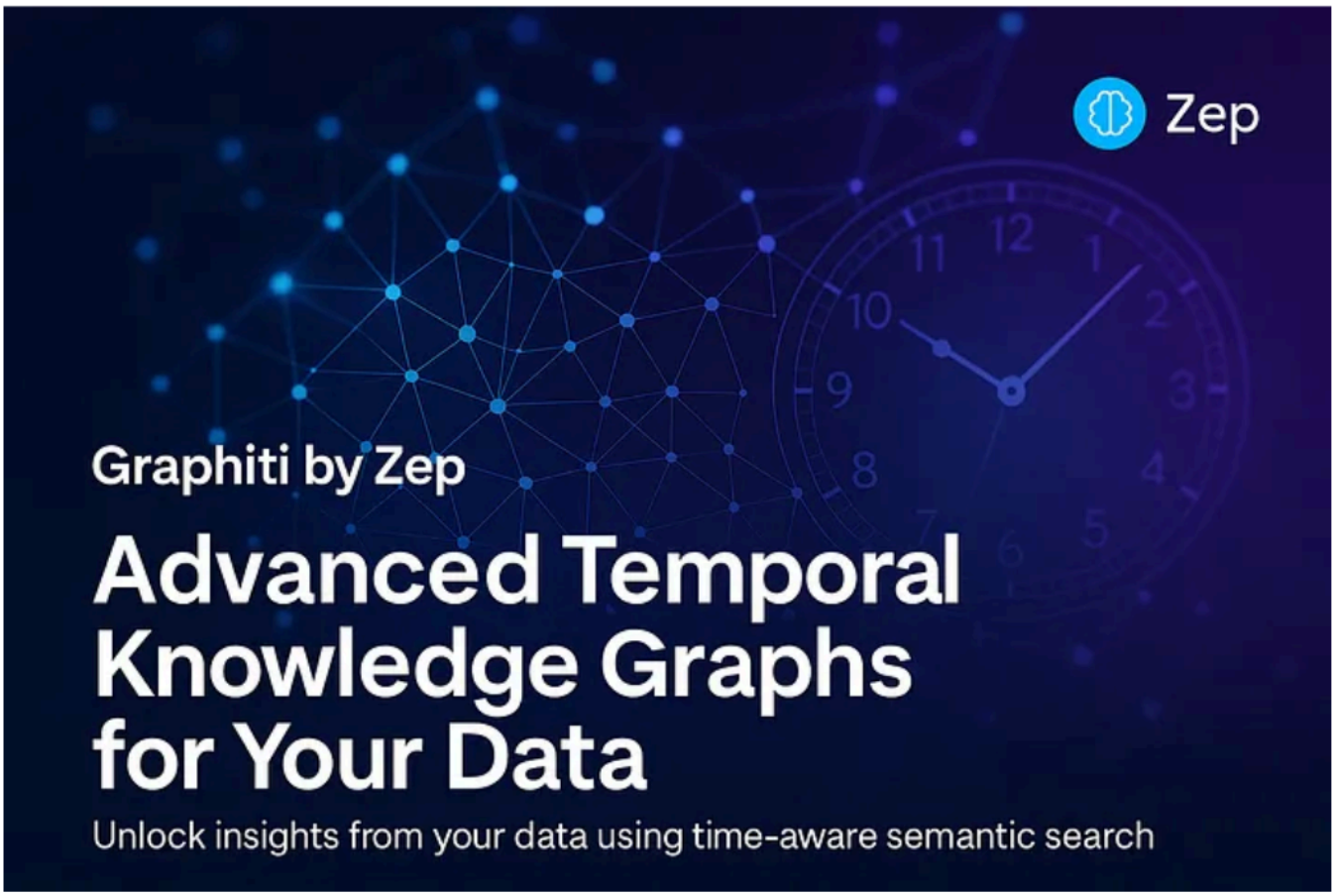
KNOWLEDGE GRAPHS | GRAPH DATABASES | TEMPORAL

Graphiti by Zep: Advanced Temporal Knowledge Graphs for Your Data

A Framework to Build, Query, Time-Aware Knowledge Graphs that work on episodic inputs

14 min read · May 14, 2025

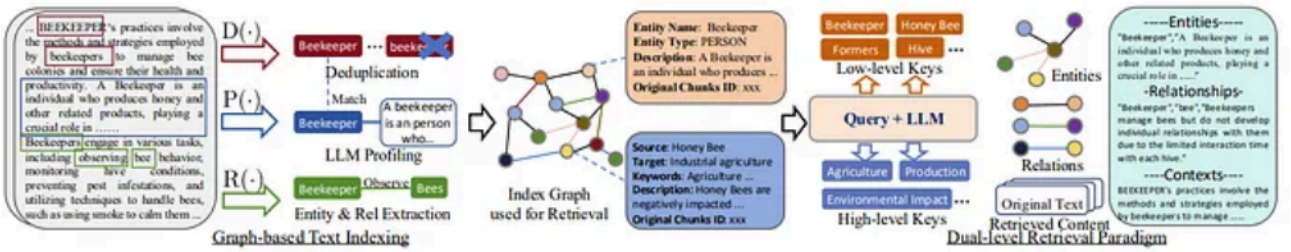
Chinmay Bhalerao



Retrieval-Augmented Generation (RAG) is a very popular architecture for building question-answering systems. It augments the answer by adding external data sources and using them to answer the questions. People found it effective until they analyzed the flaws in traditional RAG systems.

If an answer is present in more than one document, and needs to be searched across all documents, ranked, and then generated, traditional RAG fails to handle this effectively. At most, it can stack or concatenate the answers, but this is not what we expect when it comes to answer generation. The dots from all relevant documents should be connected in a way that combines all the meaningful chunks from the desired knowledge sources.

Here, knowledge graphs come into the picture. As the name suggests, knowledge graphs are structures consisting of nodes and edges, where nodes are connected to relevant other nodes in such a way that pulling one node retrieves all related knowledge on that topic. Famous architectures like Graph RAG and Light RAG use knowledge graphs to connect data effectively.



Source : Official LightRAG paper

This graph-building technique uses static data to build the graph once. When new data arrives, the entire graph is recomputed and rebuilt from scratch with all the data again. This makes the graph creation process expensive and time-consuming. GraphRAG requires the whole graph to be rebuilt whenever new data is added, which is inefficient.

In contrast, LightRAG supports dynamic environments with incremental data updates, avoiding the need to recompute everything. However, it lacks the capability to access temporal-aware knowledge.

Graphiti vs. GraphRAG

Aspect	GraphRAG	Graphiti
Primary Use	Static document summarization	Dynamic data management
Data Handling	Batch-oriented processing	Continuous, incremental updates
Knowledge Structure	Entity clusters & community summaries	Episodic data, semantic entities, communities
Retrieval Method	Sequential LLM summarization	Hybrid semantic, keyword, and graph-based search
Adaptability	Low	High
Temporal Handling	Basic timestamp tracking	Explicit bi-temporal tracking
Contradiction Handling	LLM-driven summarization judgments	Temporal edge invalidation
Query Latency	Seconds to tens of seconds	Typically sub-second latency
Custom Entity Types	No	Yes, customizable
Scalability	Moderate	High, optimized for large datasets

Source: Official Graphiti repository

Therefore, we want a lightweight framework, supports incremental data updates without full recomputation, is temporal-aware, and remains efficient for retrieval.

Here comes Graphiti to solve all problems !!!

According to **Graphiti** repository,

> *Graphiti is a framework for **building** and **querying temporally-aware knowledge graphs**, specifically tailored for AI agents operating in dynamic environments. Unlike traditional retrieval-augmented generation (RAG) methods, Graphiti **continuously integrates user interactions, structured** and **unstructured** enterprise data, and external information into a coherent, queryable graph. The framework supports **incremental data updates**, **efficient retrieval**, and **precise historical queries** without requiring complete graph recomputation, making it suitable for developing interactive, context-aware AI applications.*

Let's understand the meaning of a few concepts step by step:

> **Temporally-aware**

Temporally-aware means being **conscious of time**, specifically, being able to understand, track, and reason about how things **change over time.**

For example:

> *"Chinmay worked at Google from **2018 to 2021.**"*
>
> *"Chinmay moved to Mumbai in **March 2024.**"*

So, it **understands the sequence of events.**

So it knows:

> *Event A happened **before** Event B*
>
> *A user's preference changed over time*

It can answer time-specific queries

Like:

> *"Where did Chinmay live in 2023?"*
>
> *"What was the project status two weeks ago?"*

This temporal data model helps it to

> *Adapt to changes in user behavior or world facts*
>
> *Avoid using **outdated** information*
>
> *Remembering not just* what *happened, but* when *and* in what order

Graphiti builds **knowledge graphs** that are not just static representations of entities and relationships, but are also **temporally aware.** It tracks **how knowledge changes over time.**

Suppose you have a user named **Chinmay**, and the system sees these events over time:

1. **2022** – Chinmay lives in **Pune**
2. **2024** – Chinmay moves to **Nashik**
3. **2025** – Chinmay works at **Company X**

A regular (non-temporal) knowledge graph might just say:

```
Chinmay –[lives_in]→ Pune

Chinmay –[works_at]→ Company X
```

But a **temporally-aware knowledge graph** (like what Graphiti builds) would store:

```
Chinmay –[lived_in (2022-2024)]→ Pune

Chinmay –[lives_in (2024-)]→ Nashik

Chinmay –[works_at (2025-)]→ Company X
```

and stores **when** each fact or event was **valid or relevant.**

I hope you understood about the temporal data and its importance in graph building. The next important concept in Graphiti is Episodes.

Step 1: Create Episodic data

Converting data into episodes is the first step in the Graphiti pipeline. When we have loads of data that is beyond the context window of LLM, or we don't want to send all data unnecessarily again and again to LLM to save costs, we use chunking in that case.

In Graphiti, **episodes are chunks**, but they have important meanings.

Feature	Plain Chunk	Episode in Graphiti
Content	Raw paragraph	Sentence or paragraph (event/statement)
Time Awareness	❌ Usually none	✅ Timestamped or time-inferred
Source Tracking	❌ Optional	✅ PDF name, page, chat ID, etc.
Purpose	Splitting	Memory & graph node creation
Meaning	Arbitrary	Represents an **event** or **observation**

Why is it called "Episode"?

The term "**episodic**" comes from **Episodic Memory** in cognitive science. **Episodic Memory** is the memory of personal experiences, with time and context.

Episodes are **semantically meaningful chunks** of text. Each is treated as a **discrete memory unit** that captures a specific event or idea, along with **time and source context.**

> Episode = Chunk + Time + Meaning

To understand it better, **episodes are chunks**, but they're **specialized, time-aware chunks** designed for memory, reasoning, and graph-building.

Conversion of a normal chunk into episodes

How to Convert Normal Chunks into Episodes

Step 1: Entity and Event Extraction

Use NLP techniques to detect:

Entities (e.g., people, places, things)

Events or actions (e.g., meeting, surgery, purchase)

Timestamps or temporal phrases (e.g., “on March 3”, “last week”)

Example Chunk:

> *“John attended a meeting with the legal team on March 3, 2023, and submitted the final draft two days later.”*

Extracted:

- **Entities:** John, legal team, final draft
- **Events:** attended meeting, submitted draft
- **Time:** March 3, 2023; March 5, 2023

Step 4: Group into an Episode

- Group all events that are:
- Temporally close
- Related by topic or participant
- Define the start and end times of the episode

Final Episode:

> **Title:** *John’s Legal Review Cycle*
> **Start:** March 3, 2023
> **End:** March 5, 2023
> **Nodes:** attended meeting, submitted draft
> **Edges:** temporal sequence, participant links

While **Graphiti** can handle **normal chunks**, its true power lies in organizing and converting these chunks into **episodes** that maintain a logical structure, often with temporal awareness. If you are working with **normal chunks** only (without the need for an episodic narrative), Graphiti can still be useful, but it might not offer full support for **time-based transitions** or **contextual structuring** without shifting to episodic modeling.

Step 2: Entity & Relationship and temporal information Extraction

Once the **episodes** are created, the next step is to **extract meaningful entities** and **relationships** from the episode content.

1. Named Entity Recognition (NER):

Entities refer to meaningful objects or concepts in the text (like people, places, organizations, topics, etc.).

For example, in the episode:

> *"On March 3rd, the team finalized the rollout plan. Chinmay approved the budget."*

The entities extracted would be:

> ***Chinmay*** *(Person)*
>
> ***team*** *(Organization)*
>
> ***rollout plan*** *(Project)*
>
> ***budget*** *(Concept/Asset)*

2. Relationship Extraction:

It identifies how the entities are related in the context of the episode.

For example:

> *Chinmay approved the budget*
>
> *Team finalized rollout plan*

3. Temporal Information Extraction

It is the extraction of the information or data related to time in a chunk or episode. It will take out all references to time and use it to put the reference with respect to the time.

Step 3: Temporal Normalization

After extracting entities and relationships, the **timestamps** (dates) need to be normalized and structured. This ensures that all events and actions in the graph are temporally aligned.

1. Extract Dates:

Recognize temporal expressions (e.g., "March 3rd", "last quarter", "2025"). Parse and standardize these into a proper **timestamp format** (`yyyy-mm-dd`).

2. Create Temporal Linkage:

Every **episode** and **relationship** gets a **timestamp**, so that Graphiti can build a timeline.

Example: `"On March 3rd, Sarah approved the budget"` becomes `2025-03-03`.

Source : Official Graphiti repository

4. Fact creation

Fact creation is an important bridge between raw text and the knowledge graph structure. It is the process of extracting **structured triples** from the text.

A **fact** is a structured, atomic unit of knowledge that expresses something **true (or believed to be true)** about the world.

In a knowledge graph, a **fact** typically looks like a **triple**:

> **(subject, predicate, object)**

This is the core of **semantic representation**.

Facts are the **truth units** in a knowledge graph. They express who did what, when, where, and how — using structured triples.

> *"In June 2022, Acme Corp acquired Beta Ltd. John Smith, the CEO of Acme Corp, said this acquisition would strengthen their presence in Europe."*

```
<Subject> — <Relation> — <Object>
```

It transforms messy natural language into clean, linkable graph components. It also allows downstream reasoning systems to *understand* and *connect* disparate events.

5. Create Nodes and Edges

Many people get confused between nodes, entities, relationships, and edges. Let's clear all concepts first.

> **ENTITIES VS NODES**

Entities: These are the *concepts* or *objects* in your data. They represent real-world things or ideas (people, places, events, concepts) that you want to track in the knowledge graph.

Examples:

> ***Person**: Sarah, Dr. Rao*
>
> ***Asset**: Budget, Rollout Plan*
>
> ***Organization**: The Team, Company X*

Nodes: These are the *graph representation* of entities. In a graph, **nodes** are the **points** that represent the entities, and they connect through edges to show relationships between them. So, **nodes** are how entities are represented in a graph structure.

> **RELATIONSHIP VS EDGES**

Relationships: These describe the **connections** between entities. They tell you how two entities are related or how one entity **interacts with** or **affects** another entity.

Examples:

Sarah approved the budget (the relationship between Sarah and the budget). **The team finalized the rollout plan** (the relationship between the team and the rollout plan)

Edges: These are the **graph representation** of relationships. In a graph, **edges** are the **lines** or **arrows** that connect nodes. They represent the relationships between the entities (nodes). Edges typically have a **direction** (who did what) and sometimes additional attributes, such as **time** or **intensity**. So, **edges** are how relationships are represented in the graph.

For the Creation of Nodes and edges from the **entities** and **relationships** extracted, we create:

> *Nodes (Entities):*
> *Each **entity** becomes a **node** in the graph.*
>
> ***Sarah, budget, project** → These are all **nodes**.*
>
> *Edges (Relationships):*
> *Each **relationship** forms an **edge** between two **nodes**.*
>
> ***Sarah** → **approved** → **budget** → Relationship edge with a temporal context.*

Each **episode** and the **relationships** between nodes are **timestamped.** This temporal information adds **time-based context** to the graph, allowing you to track events over time.

> *Temporal Data on Nodes (Entities)*

Entities (nodes) in your knowledge graph represent things like people, places, objects, or events. Time normalization ensures that **each entity has a defined temporal scope** — the time period during which the entity is relevant or active.

Nodes Representing Entities:
Each node can have **temporal metadata** that describes when it was relevant or active. For instance, consider a node representing a person, **Sarah.** If Sarah held a certain role (e.g., “Manager”), you might attach temporal information to that node to indicate the time range during which she held that role.

Example:

> *Node for **Sarah** (Manager role):*

> *[Sarah] -> { "role": "Manager", "start_time": "2025-03-01T00:00:00", "end_time": "2025-06-01T00:00:00" }*
>
> ***Temporal Meaning**: This means **Sarah** was a **Manager** from **March 1st, 2025** to **June 1st, 2025**. The role is temporal and will not apply before or after that period.*

Storage of Temporal Information

For nodes:

```
{
 “id”: “Sarah”,
 “type”: “Person”,
 “role”: “Manager”,
 “start_time”: “2023-01-01”,
 “end_time”: “2023-12-31”
}
```

For edges

```
{
 “source”: “Sarah”,
 “target”: “ProjectX”,
 “relation”: “worked_on”,
 “timestamp”: “2023-03-15”
}
```

This is the approach Graphiti follows. Temporal awareness comes from **metadata attached to nodes/edges**, not from separate temporal nodes or links. A relationship (or **edge**) connects two nodes and can carry **rich metadata** that describes **how**, **when**, and **why** that relationship exists.

Typical metadata on an edge:

> *`timestamp` — When the relationship occurred*
>
> *`start_time` , `end_time` — Time range for when it's valid*

> `source` — *Original source of the information (e.g., which PDF, which sentence)*
>
> `confidence_score` — *How confident the system is in this relation*
>
> `context` — *The text snippet or episode that triggered this edge creation*
>
> `type` — *Optional sub-type of the relationship (e.g., "approved_by" vs "assigned_to")*

Relationships (edges) in Graphiti:

> *Do have* ***metadata***
>
> *Include* ***temporal information***
>
> *Help make the graph* ***temporally aware and queryable***

When is temporal data getting attached?

Temporal Metadata Attachment Stage:

During Entity + Relationship Extraction

Temporal metadata is attached at the same time entities and relationships are extracted from episodes.

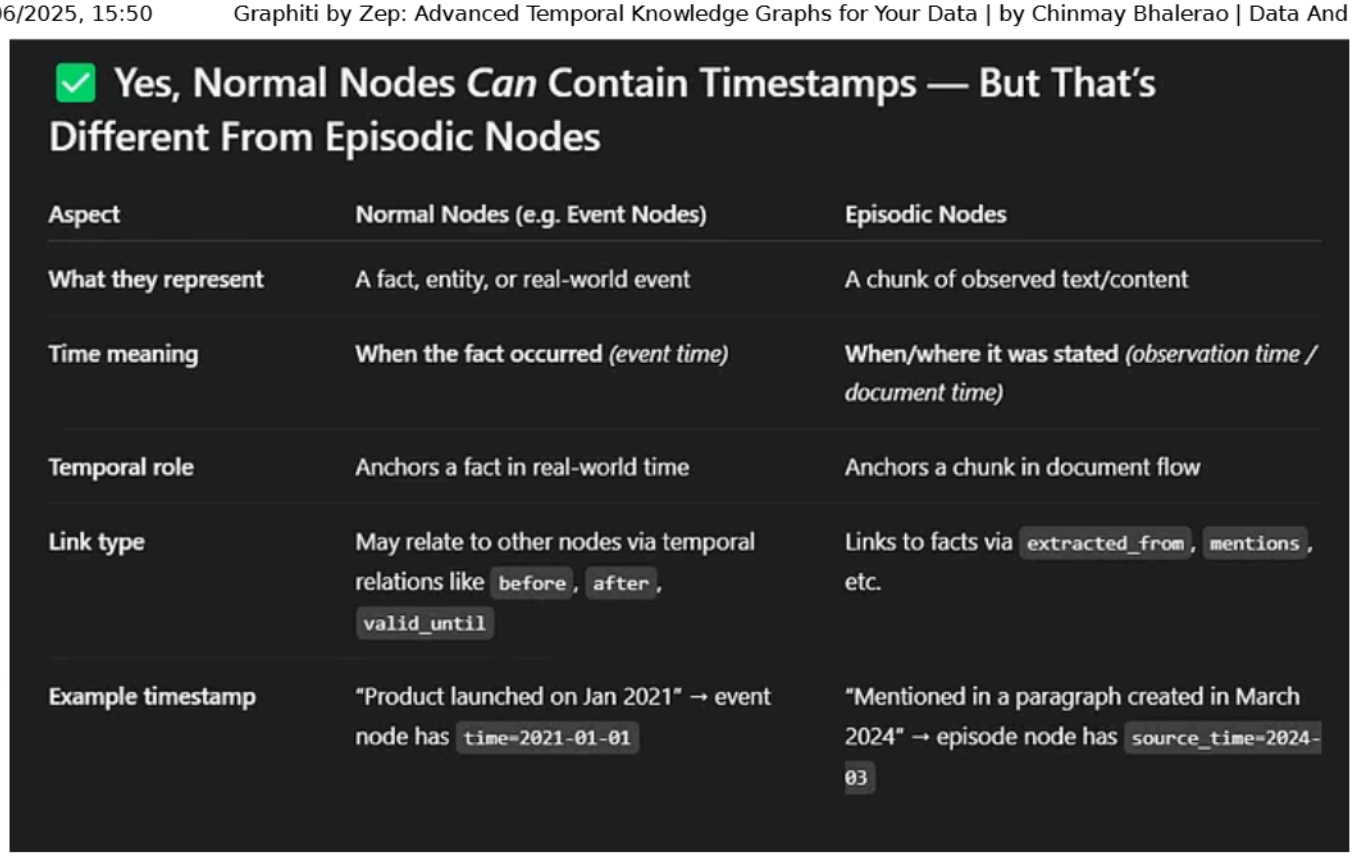
This is where:

> *Timestamps are parsed*
>
> *Time is normalized*
>
> *Metadata fields are populated for both nodes and edges*

6. CREATION OF EPISODIC NODES

Episodic nodes are not duplicates of entity/event nodes, they are **context holders** that anchor *when, where, and how* the information was observed.

Think of them like **containers** for statements or memories, not the facts themselves.



✅ Yes, Normal Nodes *Can* Contain Timestamps — But That's Different From Episodic Nodes

Aspect	Normal Nodes (e.g. Event Nodes)	Episodic Nodes
What they represent	A fact, entity, or real-world event	A chunk of observed text/content
Time meaning	**When the fact occurred** *(event time)*	**When/where it was stated** *(observation time / document time)*
Temporal role	Anchors a fact in real-world time	Anchors a chunk in document flow
Link type	May relate to other nodes via temporal relations like `before`, `after`, `valid_until`	Links to facts via `extracted_from`, `mentions`, etc.
Example timestamp	"Product launched on Jan 2021" → event node has `time=2021-01-01`	"Mentioned in a paragraph created in March 2024" → episode node has `source_time=2024-03`

One Episode → One Episodic Node

When an **episode** is created (a semantically coherent chunk of text from your source, like a paragraph or section), a **corresponding episodic node** is created in the knowledge graph.

Episodic Node Holds Relationships to All Extracted Nodes

From that chunk, when entities, events, and relationships are extracted, they are all **linked back** to the episodic node using edges like:

```
MENTIONS

CONTAINS

EXTRACTED_FROM

DESCRIBES
```

This forms a **context graph** around each episode.

> *Example Flow:*

Say you have a PDF page like:

> *"In 2022, Acme Corp acquired Beta Ltd. The CEO of Acme at the time was John Smith."*

Here's how Graphiti handles it:

Episodic Node: `Episode_42`

Metadata:

```
source_file = report.pdf

page_number = 3

chunk_id = 42

timestamp = 2023-01-01
```
(date when doc was written or observed)

Extracted Nodes:

Entity: `Acme Corp`

Entity: `Beta Ltd`

Event: `Acquisition Event`

```
event_time = 2022
```

Entity: `John Smith`

Role: `CEO`

Links from `Episode_42` **:**

```
Episode_42 → MENTIONS → Acme Corp

Episode_42 → MENTIONS → Beta Ltd

Episode_42 → CONTAINS → Acquisition Event

Episode_42 → MENTIONS → John Smith

Episode_42 → STATES_ROLE → John Smith as CEO of Acme Corp
```

Why This Matters?

You can **reconstruct the full context** around a claim. Enables **provenance tracking**: "Where was this fact said?" Supports **time-aware queries**: "What did the world look like as of this episode?"

> Each **episode** → creates one **episodic node**
> That node links to **all extracted entities, events, and facts**
> It acts as a **contextual anchor** in the knowledge graph

7. Node Deduplication & Linking

1. Check for Existing Entities (Nodes)

For each extracted entity:

> *Does a similar node already exist?*
>
> *Uses techniques like Name matching (case-insensitive), Embedding similarity (e.g., via sentence-transformers or LLM), Coreference resolution (e.g., "she" = "Sarah")*
>
> *If yes → link to the existing node*
>
> *If no → create a **new node***

2. Check for Existing Relationships (Edges)

For each proposed relationship:

> *Do the same two nodes already have this kind of edge?*
>
> *Is there already an `approved` edge between "Sarah" and "Annual Budget"?*
>
> *If yes → maybe **update timestamp**, or **add context/source***
>
> *If no → create a **new edge***

3. Merge or Add

Entities that match → are merged

Relationships that duplicate → are updated, not duplicated

This ensures a **clean, non-redundant, semantically accurate** graph. The system checks if a node (e.g., "Chinmay") already exists. If yes, it **links to the existing node.** If not, it **creates a new node.**

This step uses:

> *String matching*

> *Coreference resolution*
>
> *Embedding similarity (LLM)*

This avoids creating 10 different “Chinmay” nodes when all references point to the same person.

> *Edge Attachment*

Relationships are added between nodes:

“Sarah” — [`approved` , timestamp=2025-03-03]→ “Annual Budget”

These edges carry the **temporal and source metadata** you saw earlier.

8. Graph Embedding (Neural Memory Layer)

Each **node**, **edge**, and **episodes** are embedded into high-dimensional vectors using a model (e.g., OpenAI, SentenceTransformers, or in-house encoders).

This creates a **semantic memory layer** for Fuzzy matching, Contextual retrieval, and Semantic similarity search. Think of this as giving the graph a *neural representation* to complement the symbolic one.

2. Temporal Indexing

Temporal metadata from nodes and edges is indexed to support **time-based queries**. Allows:

> *Time slicing: “Show me events from Q3 2023”*
>
> *Validity checking: “Was this relationship still valid in 2024?”*
>
> *Timeline generation per entity*

This makes the graph *time-aware in its querying capacity*.

3. Graph Storage & Access

The final graph (symbolic + vector) is stored in a backend. Often Neo4j, ArangoDB, or a custom graph store. The vector layer might be in Pinecone, Weaviate, Qdrant, etc.. Graphiti handles both layers in sync.

Here’s the exact sequence:

Raw Input (PDFs, documents)

Episode Creation (chunking with context)
At this point:

Each chunk becomes an **Episode**

A corresponding **Episodic Node** is created in memory

Metadata is attached: `source`, `timestamp`, `chunk text`, `page number`, etc.

Entity and Relationship Extraction (from within each episode)
Entities and events are pulled **from the text inside each episodic node**

Temporal Normalization

Linking Extracted Items to Episodic Nodes
Each entity and event is linked to the episodic node it was extracted
Relationship example: `extracted_from_episode → EPISODE_XYZ`

Deduplication & Linking

Knowledge Graph Creation (episodic nodes + entities + relations now become actual graph objects)

Full Graphiti Pipeline — Step by Step

1. Input Ingestion

Accepts documents like **PDFs**, transcripts, chat logs, etc.

Each document is split into **text chunks** called **episodes.**

2 Episode Creation

Each chunk becomes an **episodic node.**

Contains metadata:

> *Source (file name, page)*
>
> *Timestamp (explicit or inferred)*
>
> *Original text*

3. Entity & Event Extraction

Uses NLP to extract:

> ***Entities*** *(people, places, companies, etc.)*
>
> ***Events*** *(actions like "acquired," "said," "moved to," etc.)*

4. Fact Construction

From the extracted elements, **facts** are generated as triples:

Subject — Predicate — Object

With metadata: timestamp, source, confidence, etc.

5. Node & Relationship Creation

Facts are encoded as:

> ***Nodes*** *for entities and events*
>
> ***Edges*** *for relationships between them*
>
> *Nodes/edges are linked to* ***episodic nodes*** *to preserve context*

6. Temporal Awareness Layer

Extracts and **normalizes time info.** Time metadata is added to nodes and relationships. Enables **temporal reasoning** (e.g., before/after, valid period)

7. Deduplication & Graph Merging

> *Prevents duplicate nodes/entities*
>
> *Links facts about the same entity across episodes/documents*

8. Graph Construction

> *Builds/updates the* ***single unified knowledge graph***
>
> *Every episode enriches the graph*

9. Query Layer (Optional)

You can now query:

> *Entity timelines*

> *Event sequences*
>
> *Episode facts*
>
> *Temporal trends*
>
> *Provenance for each fact*

Community nodes are **optional enhancements** built **after the initial graph** is stable and deduplicated.

Let's conclude everything in below flow chart.

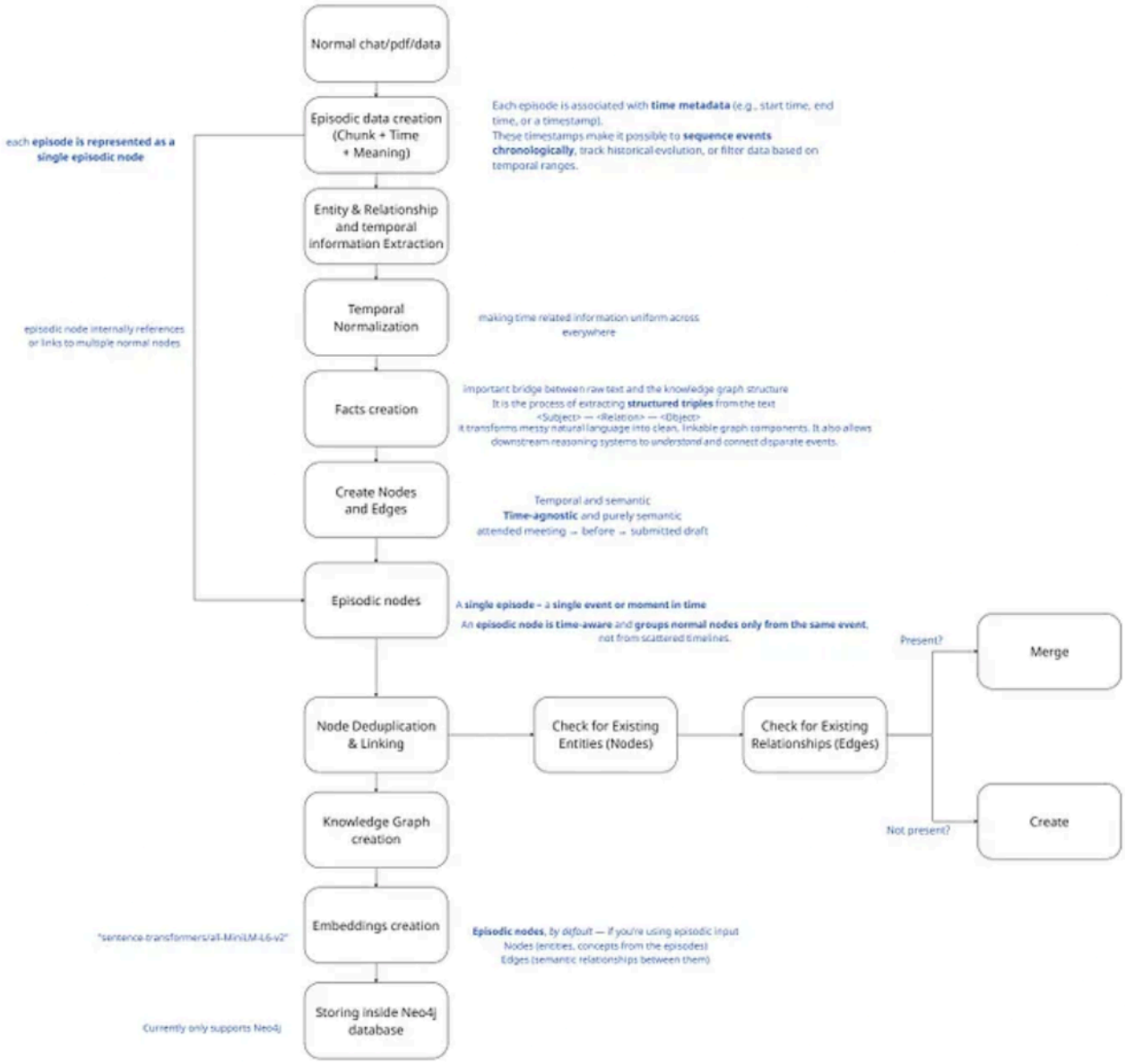


Image by Chinmay

This blog provides a comprehensive overview of using **Graphiti** to enable rich semantic search and retrieval capabilities for LLM-powered applications. We've covered the conceptual and functional aspects in detail to give you a clear understanding of how Graphiti operates in tandem with memory stores to generate contextually grounded answers.

While the complete implementation code — including data ingestion, retrieval logic, and LLM prompting — would greatly enhance your practical understanding, including it in this blog would significantly increase its length and make it harder to follow. To keep things concise and focused, we've limited the code examples here.

However, if you're interested in seeing the **end-to-end implementation** of Graphiti with **Neo4j as the underlying graph database**, feel free to reach out or drop a comment. I'll be happy to share the full working code.

If you have found this article insightful

It is a proven fact that "**Generosity makes you a happier person**"; therefore, Give claps to the article if you liked it. If you found this article insightful, follow me on **LinkedIn** and **Medium**. You can also **subscribe** to get notified when I publish articles. Let's create a community! Thanks for your support!

You can read my other blogs related to :

Huawei's Gesture-Control Photo Sharing: An AI-based Futuristic Innovation

Revolutionizing Mobile Interactions: How AI and Gesture Recognition are Shaping the Future of Seamless Photo Sharing

medium.com

Evaluating and Monitoring LLM Agents: Tools, Metrics, and Best Practices

This blog includes the tools that you can use to monitor and assess the performance of the Agentic approach

pub.towardsai.net

Feed Forward, Backpropagation & optimizers| Deep Learning: 2

This blog explores the basics of Feed Forward, Backpropagation & optimizers, and trust me this is a simple explanation

medium.com

Mastering Tracing and Monitoring of AutoGen Agents with Microsoft PromptFlow

Explore how Microsoft PromptFlow enhances the tracing and monitoring of AutoGen agents and helps debugging and...

pub.towardsai.net

Signing off,

Chinmay!

Artificial Intelligence | LIm | AI | Graph | Data

Follow

Published in Data And Beyond

991 followers · Last published 23 hours ago

Selected stories around Data Science, Machine Learning, Artificial Intelligence, Programming, and Technology topics. Writing guide: https://medium.com/data-and-beyond/how-to-write-for-data-and-beyond-b83ff0f3813e

Follow

Written by Chinmay Bhalerao

1.6K followers · 123 following

AI/ML Researcher & Engineer | 3x Top Writer in AI, CV & Object Detection | Simplifying Tech with Insights & Simulations |

Responses (3)

Pritam Shete

What are your thoughts?

Ravindra
May 19

Interested in seeing the end to end implementation.

5 1 reply Reply

Anushka Sonawane
May 19

Well Done!

5 1 reply Reply

EliEli
3 days ago

I am interested in seeing the full code. Do you have a plan for next part with more details on particular sections and steps?

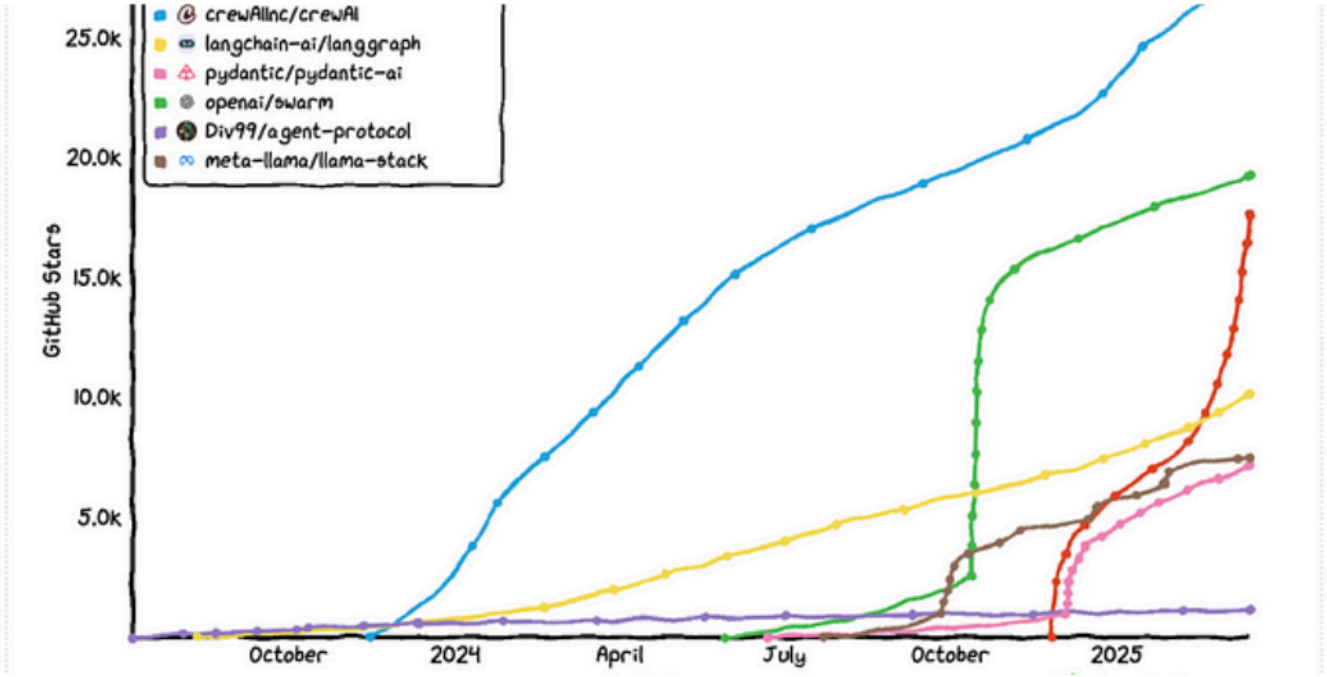
Reply

More from Chinmay Bhalerao and Data And Beyond

In Data And Beyond by Chinmay Bhalerao

Vector Databases: The Mechanics & Magic Behind Smart Data Retrieval

Jun 2 103



In Data And Beyond by TONI RAMCHANDANI

MCP Servers: A Comprehensive Guide—Another way to explain

Introduction to MCP Servers

Mar 20 339 2

In Data And Beyond by TONI RAMCHANDANI

Part 1: Introduction to n8n—What It Is and How It Works

Hey everyone, welcome to this series! Today, we're kicking off our journey into the world of automation with n8n—the one-stop workflow...

Mar 26 204 3

Chinmay Bhalerao

Converting data into SQuAD format for fine-tuning LLM models

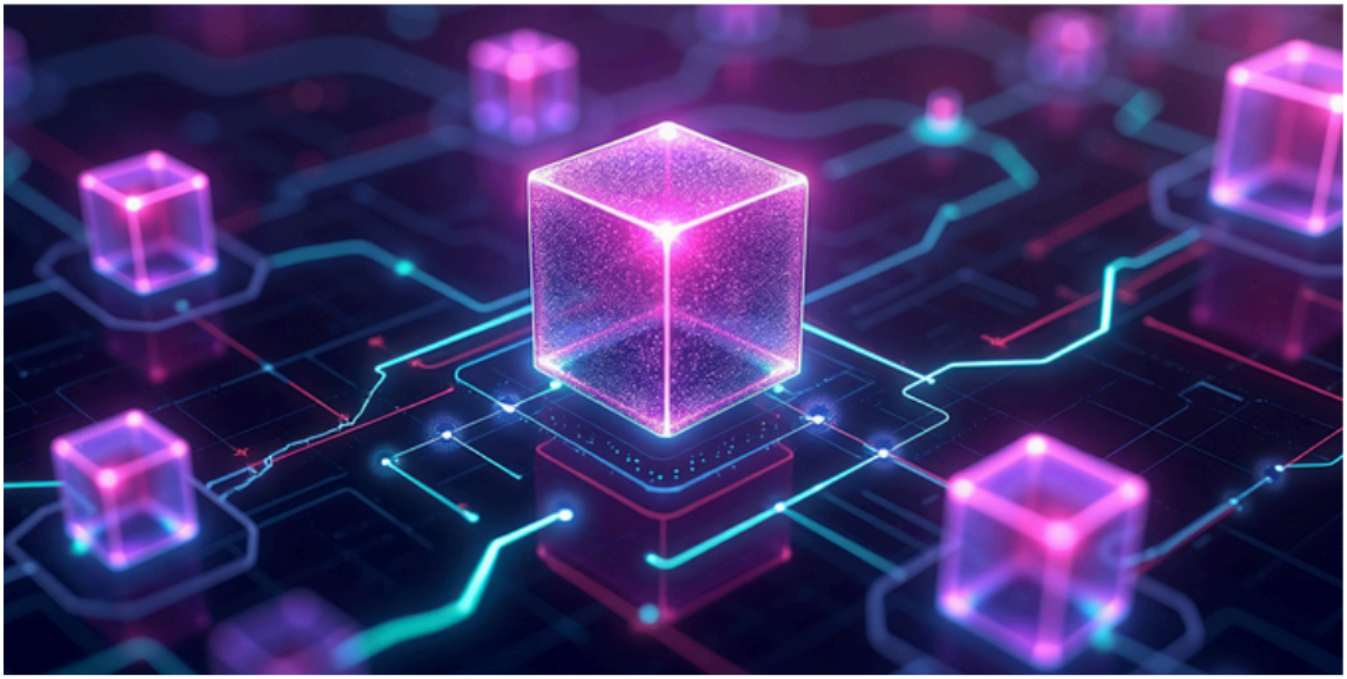
Introduction to the Haystack annotation tool and its implementation

Apr 22, 2023 172

See all from Chinmay Bhalerao

See all from Data And Beyond

Recommended from Medium

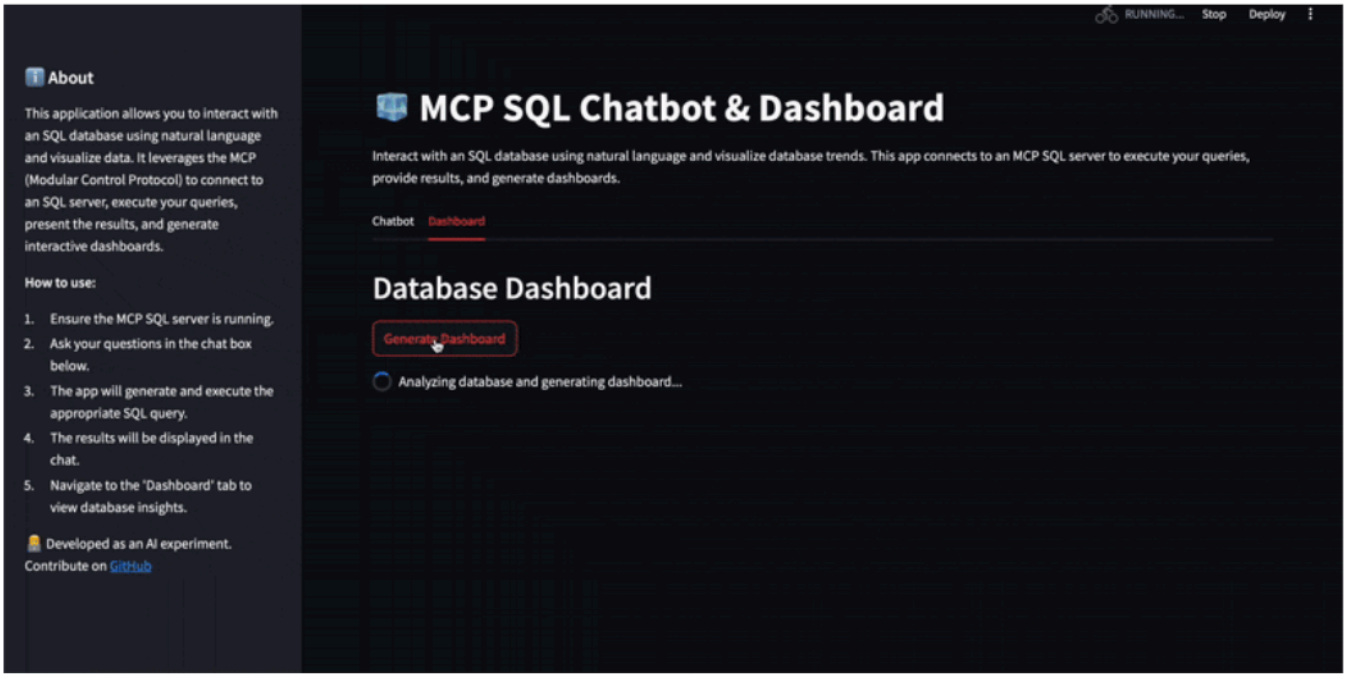


In Building Real-World, Real-Time AI by DataStax

Building Graph-Based RAG Applications Just Got Easier

Learn about a new generation of tools that simplify graph RAG. We'll walk through an example application.

May 5 38

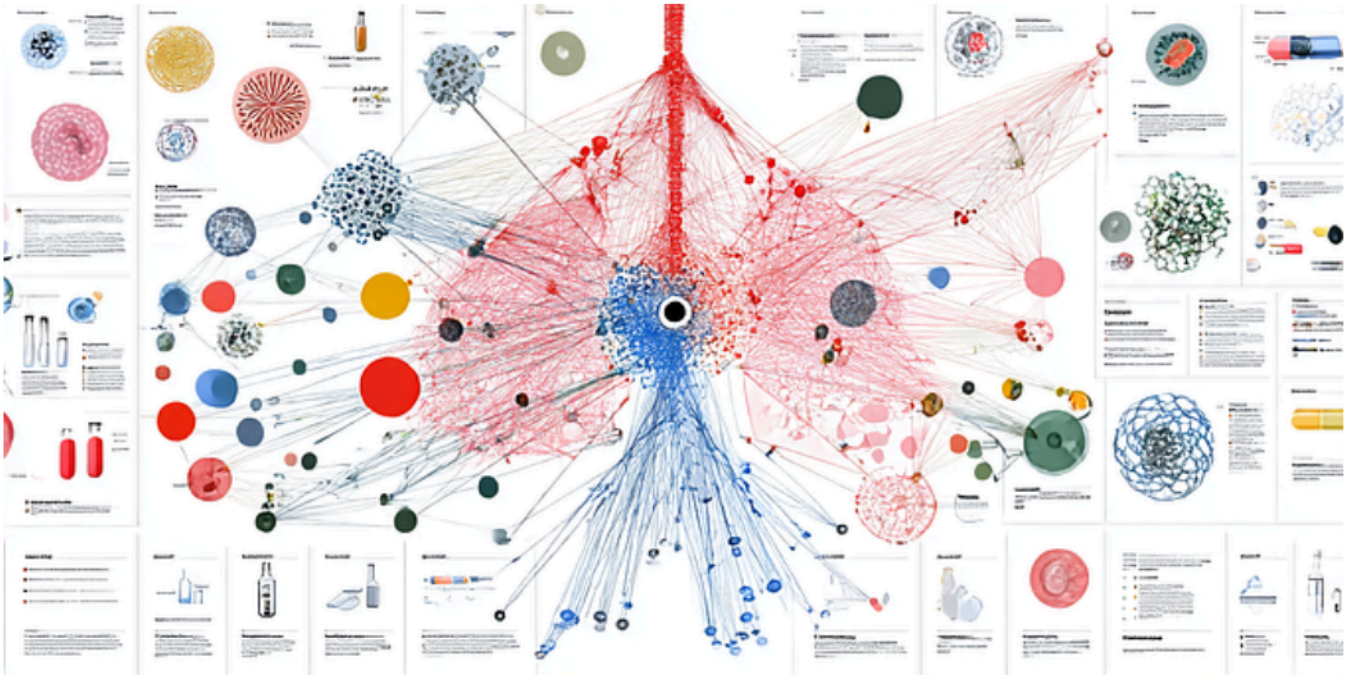


Vivek Singh Pathania

Build an AI Agent That Turns SQL Databases into Dashboards—No Queries Needed

This is a hands-on guide to building a secure, no-code, AI-powered dashboard system that connects directly to your SQL database.

Apr 17 639 14

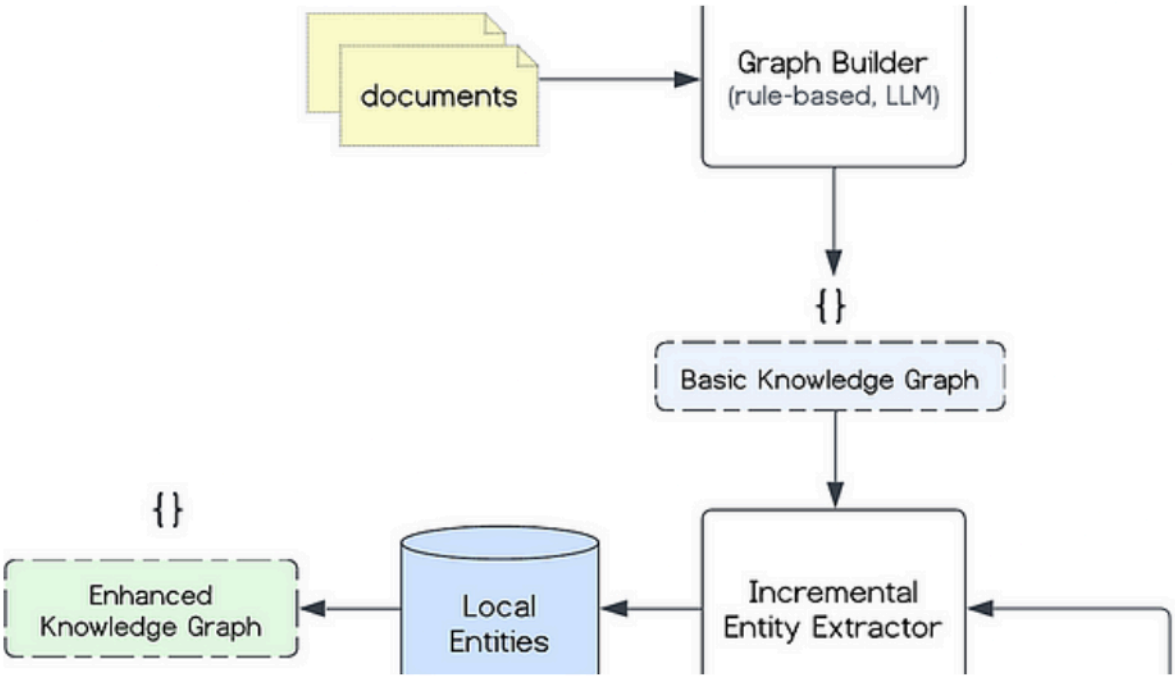


Umesh Bhatt

Latent Knowledge Graphs

Latent graphs are learned graph representations that capture implicit relationships in data without predefined structures (e.g...

May 11 101

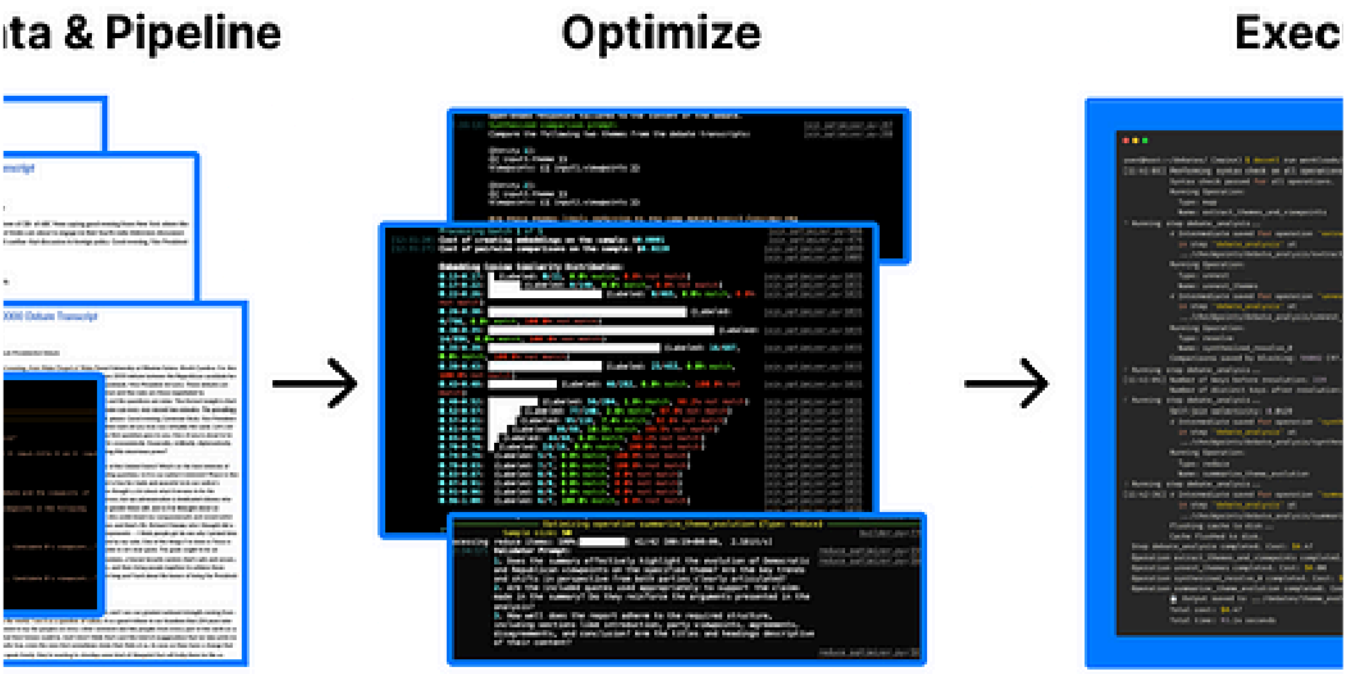


Ngoc

Part 2A: Implementing a Graph Builder to Extract a Basic Document-Centric Knowledge Graph in Python.

In this post, I'll show you how to construct a basic document-centric knowledge graph using Python and Local LLM

Jan 22 83

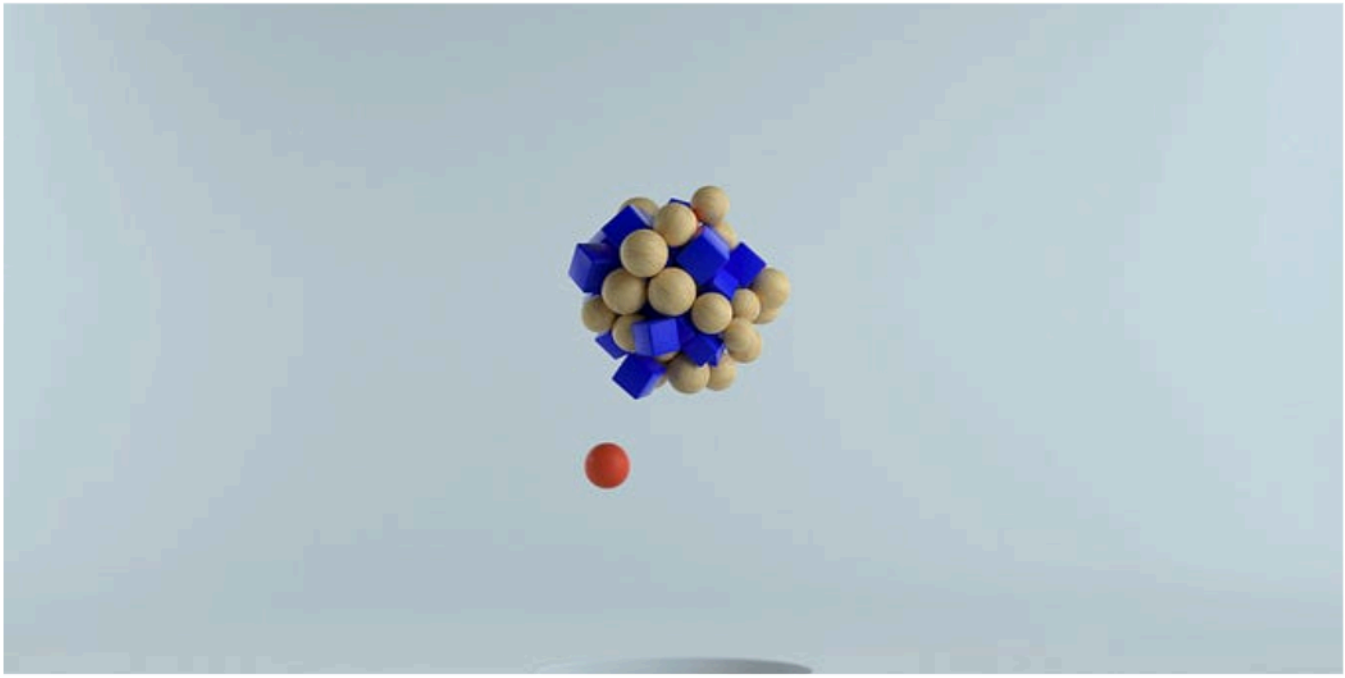


In Coding Nexus by Algo Insights

DocETL: A Game-Changing Python Library for AI-Powered Data Processing

I came across a Python library called DocETL, and I gotta say, it's pretty darn cool. I've been writing about tech for three years, and...

Apr 26 243



In Generative AI by TheMindShift

I Tried 12 AI Agent Frameworks in 2025—Here's the Brutally Honest Guide You Actually Need

From No-Code Builders to Scalable Dev Tools—Here's What Actually Works (and What Doesn't)

Jun 4 1.4K 52