Building the Next-Gen Agentic App with GraphRAG & NVIDIA cuGraph Introduction

This documentation outlines the step-by-step process for integrating graph-based AI techniques with agentic applications, leveraging GraphRAG, NVIDIA cuGraph for GPU-accelerated graph analytics, and OpenVINO for optimized inference. The workflow incorporates LangGraph, NetworkX, and ArangoDB to enhance analytics and decision-making in AI-driven applications.

Steps we have performed

Step 0: Problem Statement & Selecting the Dataset Problem Statement

In modern healthcare, effective diagnosis, treatment planning, and drug safety monitoring require efficient access to vast amounts of medical data. However, traditional medical databases store information in unstructured or tabular formats, making it challenging to:

* **Retrieve relevant disease, symptom, and treatment relationships efficiently.**
* **Identify drug interactions and potential adverse effects.**
* **Uncover genetic predispositions to diseases.**
* **Generate explainable AI-driven medical insights.**

A graph-based approach is well-suited to tackle these challenges by structuring data as interconnected entities (patients, diseases, symptoms, treatments, drugs, and genes), enabling faster and more context-aware medical analysis.

Why This Dataset?

The chosen dataset is a synthetic medical knowledge graph, designed to simulate real-world medical relationships and facilitate GraphRAG-based AI applications for improved decision-making. It is particularly suitable for solving the problem statement due to the following reasons:

1. **Graph Structure for Real-Time Insights:**
   * **The dataset is structured as a knowledge graph, allowing AI models to retrieve relevant disease, symptom, and treatment relations in real time.**
2. **Explainability & Trustworthy AI Decisions:**
   * **Medical decisions require transparency. The dataset enables AI models to provide query results backed by structured graph insights, improving interpretability.**
3. **Optimized Query Efficiency:**
   * **Unlike traditional unstructured medical records, a graph database structure accelerates disease and treatment information retrieval, reducing processing time.**
4. **Enhanced Predictive Capabilities:**
   * **AI-powered inference can analyze patterns in patient diagnoses, drug interactions, and genetic linkages to predict potential risks and provide personalized recommendations.**

Dataset Creation Process

To build this dataset, I followed these key steps:

1. **Synthetic Data Generation:**
   * **Used Faker to generate 10,000 synthetic patient records, ensuring a diverse dataset representative of real-world cases.**
2. **Medical Relationship Mapping:**
   * **Defined mappings between diseases, symptoms, treatments, drugs, and genes to establish meaningful medical connections.**

With this dataset prepared, I proceeded to Step 1, where I transformed it into a structured graph format for further processing and analysis.

Step 1: Preparing the Dataset for Graph-Based Analysis Why Prepare the Dataset?

Before a dataset can be used in a graph-based system, it must be structured in a way that allows connections between different entities to be clearly defined. This step is crucial because traditional data (such as CSV files or relational databases) typically stores information in a tabular format, making it difficult to analyze relationships efficiently.

By transforming the data into a graph structure, we can:

* **Understand how different medical entities (patients, diseases, drugs, etc.) interact.**
* **Identify hidden patterns, such as potential disease outbreaks, drug interactions, or hereditary disease risks.**
* **Enable AI-powered Graph Retrieval-Augmented Generation (GraphRAG) for real-time medical recommendations.**

1. **Loading the Dataset & Displaying Initial Data**

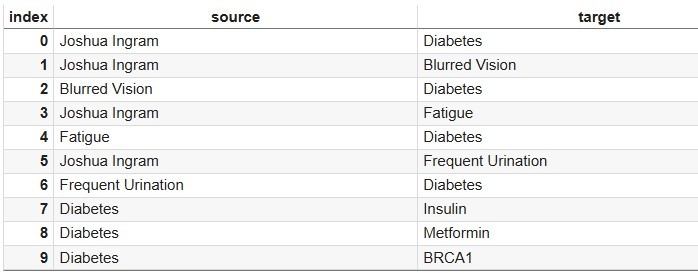
The dataset consists of medical records that capture interactions between different entities such as patients, diseases, symptoms, treatments, drugs, and genes. Each row represents a relationship between two entities.

Structure of the Data

Each record in the dataset contains three key components:

* 1. **Source: The starting entity (e.g., a patient, a disease, a drug).**
  2. **Target: The connected entity (e.g., the disease a patient is diagnosed with, the symptoms associated with a disease, or the drugs that interact).**
  3. **Relationship Type: Defines how the source and target are connected.**

Example - First 10 Rows of the Dataset:



From this table, we can observe:

* + - **Patients are diagnosed with diseases.**
    - **Symptoms are linked to diseases.**
    - **Certain diseases are treated with specific medications.**
    - **Some drugs interact with each other, which can be important for pharmaceutical safety.**
    - **Genetic markers are linked to diseases, helping identify hereditary risks.**

\*\*\*\*\*Also some EDA with the csv file(EDA)

1. **Defining Node Types (Entities in the Graph)**

In a graph-based system, nodes represent the key entities in the dataset. Each entity type is categorized based on its role in the dataset.

Types of Nodes in This Dataset:

* **Patient: Represents an individual receiving medical treatment (e.g., "Joshua Ingram").**
* **Disease: Represents a diagnosed medical condition (e.g., "Diabetes", " COVID-19").**
* **Symptom: Represents an observable medical issue (e.g., "Fever", "Cough").**
* **Treatment: Represents medical interventions used to treat diseases (e.g., " Insulin", "Beta Blockers").**
* **Drug: Represents pharmaceutical substances (e.g., "Aspirin", " Paracetamol").**
* **Gene: Represents genetic factors that may influence diseases (e.g., "BRCA1", " TP53").**

Each node has a unique identifier to distinguish it from others.

1. **Defining Edge Types (Relationships Between Entities)**

Edges in a graph represent connections or relationships between nodes. Each relationship is defined based on medical knowledge and plays a key role in understanding interactions between different entities.

Types of Relationships in This Dataset:

Relationship Type Meaning

PATIENT\_DIAGNOSED\_WITH A patient has been diagnosed with a disease. PATIENT\_HAS\_SYMPTOM A patient is experiencing a specific symptom. SYMPTOM\_INDICATES\_DISEASE A symptom is commonly associated with a specific disease. DISEASE\_HAS\_TREATMENT A disease can be treated with a specific treatment.

DISEASE\_HAS\_GENE A disease has a genetic association with a specific gene. DISEASE\_HAS\_DRUG A disease can be managed using a particular drug.

DRUG\_INTERACTS\_WITH Two drugs interact with each other (which may be harmful or

beneficial).

Example Relationships in a Graph Format:

* **"Joshua Ingram" → [PATIENT\_DIAGNOSED\_WITH] → "Diabetes"**
* **" Joshua Ingram " → [PATIENT\_HAS\_SYMPTOM] → "Fever"**
* **"Fever" → [SYMPTOM\_INDICATES\_DISEASE] → " COVID-19"**
* **" COVID-19" → [DISEASE\_HAS\_TREATMENT] → "Oxygen Therapy"**
* **"Diabetes" → [DISEASE\_HAS\_GENE] → " BRCA1"**
* **"Aspirin" → [DRUG\_INTERACTS\_WITH] → "Ibuprofen"**

These relationships allow us to visualize how different entities are connected, making it easier to draw meaningful conclusions from the dataset.

1. **Storing the Pre-processed Data for Graph Construction**

After defining node types and relationship types, I ensured the dataset was correctly formatted for the next step: graph construction.

This means:

Nodes were categorized correctly (Patient, Disease, Drug, etc.). Relationships were clearly mapped between nodes.

All entities had unique identifiers to prevent duplication.

The data was structured for easy ingestion into NetworkX in the next step.

Step 2: Loading the Graph Data into NetworkX

Why Load the Dataset into NetworkX?

Now that the dataset is structured into nodes (entities) and edges (relationships), it needs to be converted into an actual graph for analysis.

A graph-based representation allows us to:

⬛✓ Visualize medical relationships (e.g., which symptoms indicate which diseases).

✓⬛ Analyze how diseases, treatments, and genetic factors are interconnected.

✓⬛⬛ Run queries efficiently instead of searching through large tables.

⬛✓⬛ Detect patterns in patient diagnoses, treatment outcomes, and drug interactions.

1. **Initializing the Graph Structure**

To store the structured data, I created a graph object using NetworkX.

At this stage, I initialized an empty graph where nodes and edges would be added in the following steps.

1. **Adding Nodes (Entities) to the Graph**

After defining node types in Step 1, I added all entities as nodes into the graph. Logic for Adding Nodes

Each node in the graph represents a real-world medical entity, and I included extra attributes to describe them.

Node Type Example Node Additional Attributes

Patient Joshua Ingram {"type": "Patient", "id": "John Doe"} Disease Diabetes {"type": "Disease", "id": "Diabetes"} Symptom Fever {"type": "Symptom", "id": "Fever"} Treatment Beta Blockers {"type": "Treatment", "id": "Beta Blockers"} Drug Aspirin {"type": "Drug", "id": "Aspirin"}

Gene BRCA1 {"type": "Gene", "id": " BRCA1"} Why Include Attributes?

* + **The "type" attribute ensures each node is correctly categorized.**
  + **The "id" attribute ensures each node is uniquely identifiable.**

This setup makes it easier to query the graph and extract meaningful insights later.

1. **Adding Edges (Relationships) to the Graph**

Once all nodes were added, I connected them using edges based on the relationships defined in Step 1.

Logic for Adding Edges

Each edge represents a real-world medical relationship, with extra attributes describing the connection.

Relationship Type Example Edge Additional Attributes PATIENT\_DIAGNOSED\_WITH ("John Doe", "Diabetes") {"type": "PATIENT\_DIAGNOSED\_WITH"} PATIENT\_HAS\_SYMPTOM ("John Doe", "Fever") {"type": "PATIENT\_HAS\_SYMPTOM"}

SYMPTOM\_INDICATES\_DISEASE ("Fever", "COVID-19") {"type":

"SYMPTOM\_INDICATES\_DISEASE"}

DISEASE\_HAS\_TREATMENT ("COVID-19", "Oxygen

Therapy")

{"type": "DISEASE\_HAS\_TREATMENT"}

DISEASE\_HAS\_GENE ("Diabetes", "BRCA1") {"type": "DISEASE\_HAS\_GENE"}

DRUG\_INTERACTS\_WITH ("Aspirin", "Ibuprofen") {"type": "DRUG\_INTERACTS\_WITH"} Why Include Attributes?

* **The "type" attribute helps identify different relationship categories in the graph.**
* **These attributes allow for faster filtering and retrieval of specific connections.**

Now, the graph has both nodes (entities) and edges (relationships), making it a fully connected medical knowledge graph.

1. **Verifying the Graph Structure**

After constructing the graph, basic verification steps were performed: Graph Summary

* + **Total number of nodes: 9409**
  + **Total number of edges: 197116 Sample Nodes:**

('medical\_node/7581', {'\_id': 'medical\_node/7581', '\_key': '7581', 'id': 'Allen Chang', 'type': 'Patient'}),

('medical\_node/1454', {'\_id': 'medical\_node/1454', '\_key': '1454', 'id': 'Lisa Miller', 'type': 'Patient'}),

('medical\_node/8422', {'\_id': 'medical\_node/8422', '\_key': '8422', 'id': 'Tommy Daniels', 'type': 'Patient'}),

('medical\_node/6085', {'\_id': 'medical\_node/6085', '\_key': '6085', 'id': 'Jennifer Fuller', 'type': 'Patient'}),

('medical\_node/1407', {'\_id': 'medical\_node/1407', '\_key': '1407', 'id': 'Robert Merritt', 'type': 'Patient'})

Sample Edges:

('medical\_node/8331', 'medical\_node/86', {'\_from': 'medical\_node/86', '\_id': 'medical\_node\_to\_medical\_node/196195', '\_key': '196195', '\_to': 'medical\_node/8331', 'type': 'PATIENT\_HAS\_SYMPTOM'}),

('medical\_node/22', 'medical\_node/7241', {'\_from': 'medical\_node/22', '\_id': 'medical\_node\_to\_medical\_node/93953', '\_key': '93953', '\_to': 'medical\_node/7241', 'type': 'PATIENT\_DIAGNOSED\_WITH'}),

('medical\_node/22', 'medical\_node/1263', {'\_from': 'medical\_node/22', '\_id': 'medical\_node\_to\_medical\_node/89564', '\_key': '89564', '\_to': 'medical\_node/1263', 'type': 'PATIENT\_DIAGNOSED\_WITH'}),

('medical\_node/18', 'medical\_node/7832', {'\_from': 'medical\_node/18', '\_id': 'medical\_node\_to\_medical\_node/82074', '\_key': '82074', '\_to': 'medical\_node/7832', 'type': 'PATIENT\_HAS\_SYMPTOM'}),

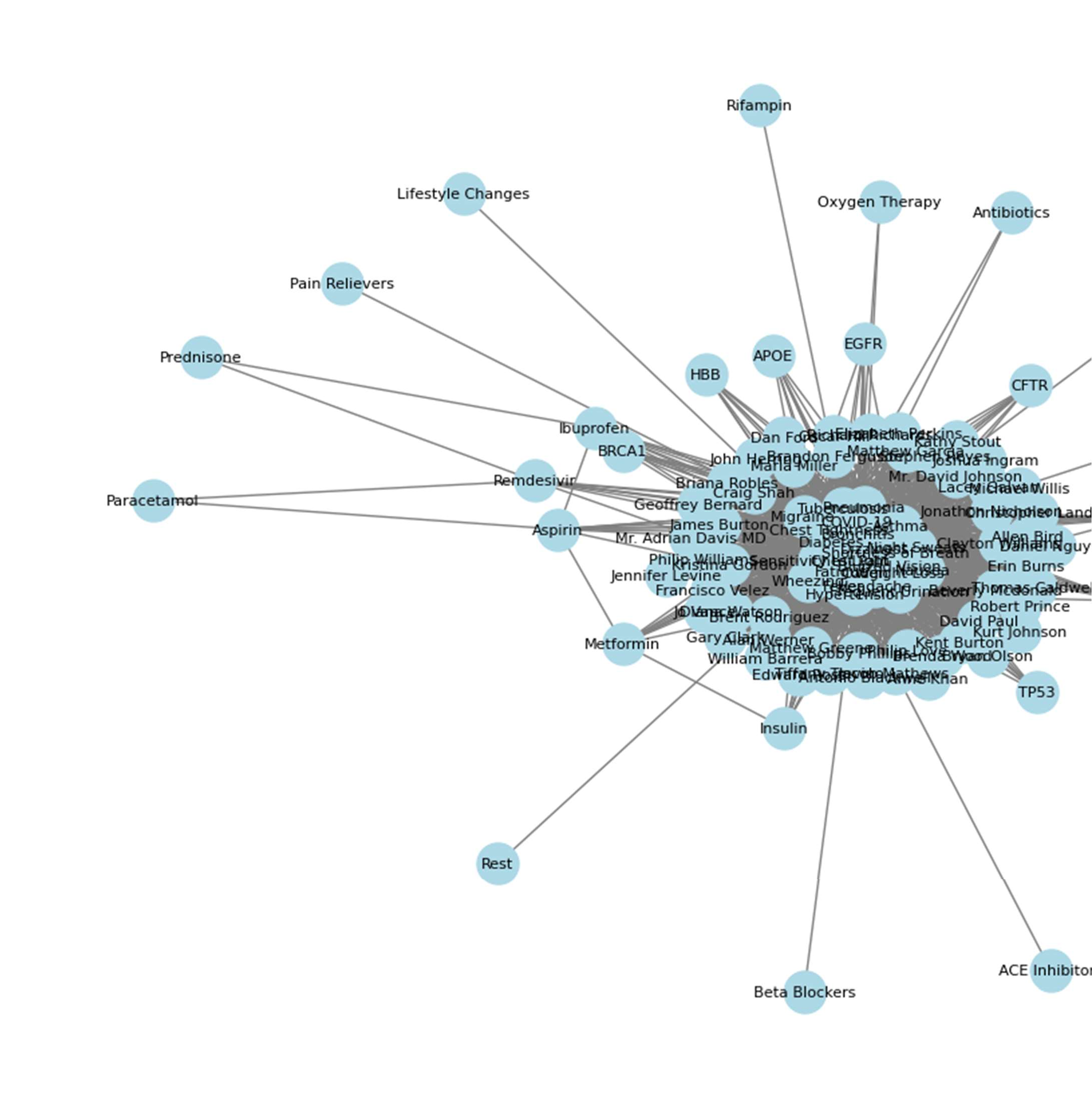
('medical\_node/5051', 'medical\_node/16', {'\_from': 'medical\_node/16', '\_id': 'medical\_node\_to\_medical\_node/61878', '\_key': '61878', '\_to': 'medical\_node/5051', 'type': 'PATIENT\_DIAGNOSED\_WITH'})

This verification ensures that all entities and relationships were correctly mapped.

1. **Visualizing the Graph**

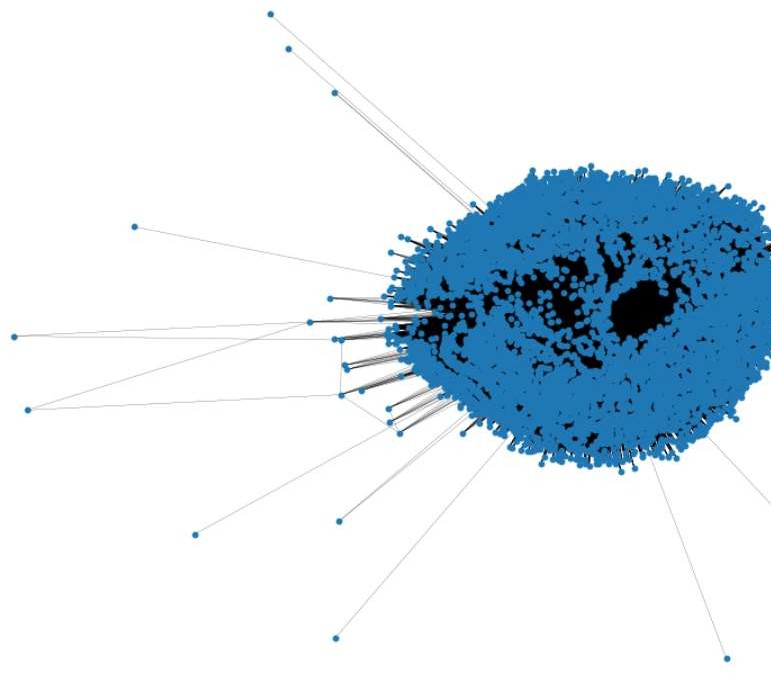
To better understand the structure of the dataset, a subset of the graph (100 nodes) was visualized.

Graph Visualization:



Graph visualization showing a sample of the medical knowledge graph.

Also a visualisation of the full graph is displayed below



The Full Graph Visualisation

Step 3: Persist / Connect the Graph in ArangoDB

Step 3: Persisting and Connecting the Graph in ArangoDB

Why Store the Graph in ArangoDB?

After successfully constructing the medical knowledge graph in NetworkX, the next step is to store it in a graph database for scalability, efficient querying, and integration with advanced analytics tools.

ArangoDB is chosen because:

⬛✓ It is a multi-model graph database, allowing storage of complex relationships.

⬛✓ It enables efficient querying of graph data, making it ideal for analyzing patient-disease interactions, drug interactions, and genetic links.

✓⬛ It integrates seamlessly with NetworkX, allowing for fast analytics and easy CRUD operations.

⬛✓ It supports GPU-accelerated analytics via cuGraph, enabling high-performance processing of large-scale graphs.

1. **Connecting to ArangoDB**

Since the graph is currently stored in NetworkX, it needs to be persisted to ArangoDB. To establish a connection with ArangoDB, the ArangoClient was used.

Connecting to ArangoDB



* + **hosts: Specifies the ArangoDB cloud instance URL.**
  + **username and password: Used to authenticate access to the database.**
  + **verify=True: Ensures secure communication with the ArangoDB instance.**

Once connected, the next step is to store the NetworkX graph into ArangoDB.

1. **Storing the Graph in ArangoDB Using NetworkX Integration**

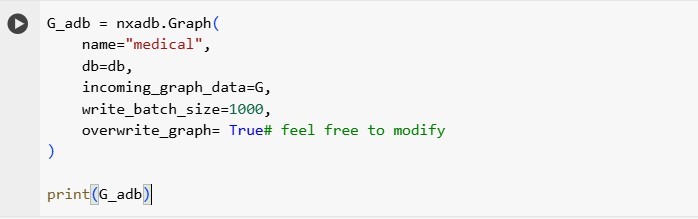
The ArangoDB NetworkX adapter (nxadb) was used to transfer the graph from NetworkX to ArangoDB.

Step 2.1: Create and Store the Graph in ArangoDB

To store the NetworkX graph in ArangoDB, the following process was used:

1. **A new graph named "medical" was created in ArangoDB.**
2. **The existing NetworkX graph (G) was transferred into ArangoDB.**
3. **A batch write process (1000 records per batch) was used to improve performance.**
4. **Existing data was overwritten to ensure the latest version was stored.**

Graph Persistence Using NetworkX-ArangoDB Adapter



1. **Understanding How Nodes and Edges Are Stored in ArangoDB**

Once persisted, ArangoDB automatically assigns generic names to both nodes and edges: Nodes:

* + **All nodes in the graph are stored under a single collection named medical\_node.**
  + **Regardless of whether the node is a Patient, Disease, Symptom, Treatment, Drug, or Gene, they are all stored in medical\_node.**

**Example Nodes Stored in ArangoDB**

**Node Type ArangoDB Collection Example Document**

**Patient** medical\_node { "\_key": "JohnDoe", "type": "Patient" } **Disease** medical\_node { "\_key": "Diabetes", "type": "Disease" } **Symptom** medical\_node { "\_key": "Fever", "type": "Symptom" }

dges:

* + **All relationships between nodes are stored in a single edge collection named medical\_node\_to\_medical\_node.**
  + **Regardless of the type of relationship (e.g., PATIENT\_DIAGNOSED\_WITH, DISEASE\_HAS\_TREATMENT), they all reside in medical\_node\_to\_medical\_node.**

Example Edges Stored in ArangoDB

Relationship Type ArangoDB Edge Collection Example Edge Document

{ "\_from": "medical\_node/JohnDoe", "\_to":

PATIENT\_DIAGNOSED\_WIT medical\_node\_to\_medical\_ "medical\_node/Diabetes",

H node

"type": "PATIENT\_DIAGNOSED\_WIT H" }

{ "\_from":

PATIENT\_HAS\_SYMPTOM medical\_node\_to\_medical\_ "medical\_node/JohnDoe", "\_to":

node

"medical\_node/Fever", "type": "PATIENT\_HAS\_SYMPTOM" }

Relationship Type ArangoDB Edge Collection Example Edge Document

{ "\_from": "medical\_node/Fever", "\_to":

SYMPTOM\_INDICATES\_DIS medical\_node\_to\_medical\_ "medical\_node/Influenza",

EASE

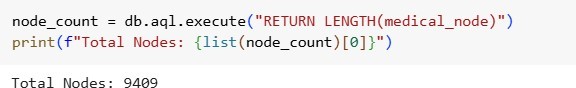
node

"type": "SYMPTOM\_INDICATES\_DISE ASE" }

1. **Validating the Graph in ArangoDB**

After inserting nodes and edges, validation queries were executed to confirm successful storage. Graph Validation Queries

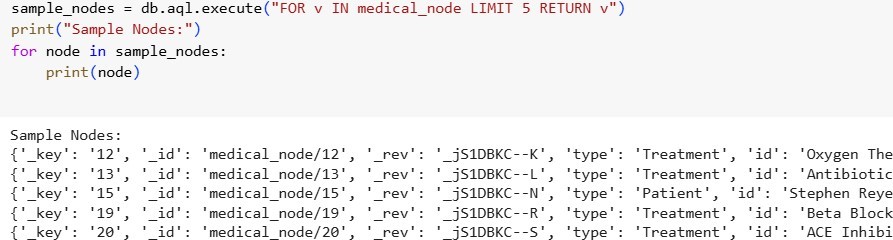
1. **Count Total Nodes in ArangoDB**



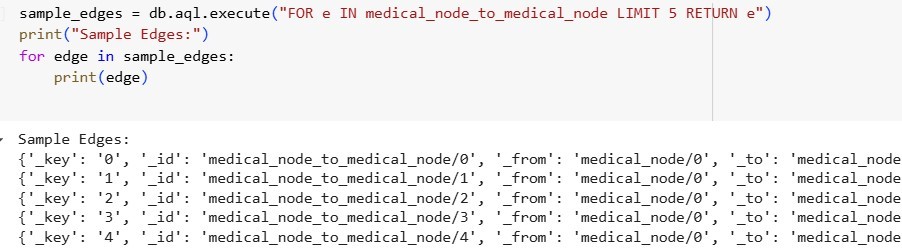
1. **Count Total Edges in ArangoDB**



1. **Sample Node Query**



1. **Sample Edge Query**



These queries confirmed that:

All nodes and relationships were correctly stored.

Data could be queried efficiently using ArangoDB AQL (ArangoDB Query Language). Why Build an Agentic App?

Now that our medical knowledge graph is fully structured and stored in ArangoDB, we need an agentic system that allows users to query and analyze the data efficiently.

The Agentic App will:

* + **Enable natural language queries that dynamically determine the best retrieval method.**
  + **Use various query executions including AQL, Networkx Algorithms, Visualisation, Subgraphing and most importantly Hybrid Query Execution, combining AQL (ArangoDB Query Language) and GPU-accelerated cuGraph analytics for complex graph queries.**
  + **Be implemented using LangGraph and AutoGen, two powerful agentic frameworks.**
  + **Feature GPT-4o (OpenAI) as the LLM to interpret user queries and generate responses.**

**Code explanation for AI agentic app with Langchain and Langraph**

The code defines various tools and functions that use **LangChain, OpenAI's GPT-4o, ArangoDB, and NetworkX** to process **graph-based queries** in a **Medical Knowledge Graph**. Below, I'll break down each key function and how it fits into the system.

**1️⃣ text\_to\_aql\_to\_text(query: str)**

**Purpose: Converts a natural language query into an ArangoDB Query Language (AQL) query.**

**How it works?**

1. **Defines allowed edge types** in the graph, such as:
   * "DISEASE\_HAS\_TREATMENT"
   * "PATIENT\_DIAGNOSED\_WITH"
   * "DISEASE\_HAS\_DRUG", etc.
2. **Constructs an instruction prompt** that asks GPT-4o to:
   * Convert a natural language query to AQL.
   * Ensure it only uses **predefined edge types** (no assumptions).
3. **Uses ChatOpenAI (GPT-4o)** to generate the AQL query.
4. **Uses ArangoGraphQAChain** to execute the query on the **ArangoDB graph**.
5. **Returns the extracted results** from the graph.

**2️⃣ text\_to\_nx\_algorithm\_to\_text(query)**

**Purpose: Converts a natural language query into NetworkX graph computations.**

**How it works?**

1. **Uses GPT-4o to generate Python NetworkX code** that:
   * Finds **disease relationships, symptoms, drug interactions, etc.**
   * Computes **graph algorithms** (e.g., shortest paths, centrality, connectivity).
2. **Executes the generated NetworkX code** dynamically using exec().
3. **Extracts the computation result** and sends it to GPT-4o for a **concise explanation**.
4. **Returns a human-readable answer**.

**3️⃣ execute\_hybrid\_query(query)**

**Purpose: Determines whether to use AQL (ArangoDB) or NetworkX (or both) based on the query.**

**How it works?**

1. **Uses GPT-4o to classify the query** into:
   * **AQL Query** (for database retrieval).
   * **NetworkX Algorithm** (for graph computation).
2. **Executes AQL if needed**:
   * Runs the query on ArangoDB and extracts relevant results.
   * Uses GPT-4o to **extract key entities** (e.g., symptoms, drugs).
3. **Executes NetworkX if needed**:
   * Applies the required **graph algorithm** (PageRank, Betweenness Centrality, etc.).
   * Extracts metric values for relevant nodes.
4. **Filters relevant nodes** based on **AQL results** and **NetworkX analysis**.
5. **GPT-4o summarizes the findings** in natural language.
6. **Returns the final response.**

**4️⃣ extract\_nx\_subgraph(query)**

**Purpose: Extracts a subgraph from the full NetworkX graph.**

**How it works?**

1. **Converts the ArangoDB graph into NetworkX (G\_nx)**.
2. **Uses GPT-4o to generate NetworkX code** that extracts only the relevant nodes/edges.
3. **Executes the generated Python code** to get the subgraph.
4. **Summarizes the extracted subgraph** (nodes, edges, sample node/edge).
5. **Returns the extracted subgraph**.

Autogen AI Agent

Our **Autogen-powered AI agent** in **ArangoMedica** acts as an **autonomous decision-maker** that analyzes user queries, selects the appropriate graph processing tool, and executes it. It integrates **ArangoDB (AQL), NetworkX, GPT-4o, and visualization tools** to provide deep insights into medical knowledge graphs.

**🔹 Step-by-Step Functioning of Our Autogen Agent**

**1️⃣ User Query Processing (Natural Language to Structured Action)**

* A user inputs a **natural language query**, such as:
  + *"Find all patients diagnosed with Diabetes."*
  + *"Identify the most influential symptom using PageRank."*
* The query is sent to our **Autogen-based AI agent**, GraphAssistant.

**2️⃣ Decision-Making: Autogen Agent Analyzes the Query**

* The agent reads a **structured system prompt** to decide **which tool should be used**.
* Using **GPT-4o**, the agent classifies the query into:
  + **AQL Execution (Database Retrieval)**
  + **NetworkX Execution (Graph Computation)**
  + **Hybrid Execution (Both AQL + NetworkX)**
  + **Subgraph Extraction**
  + **Graph Visualization**
* It **selects the best tool** based on the intent of the query.

**3️⃣ Tool Selection and Execution**

* The **selected tool** is extracted using **regex-based matching**.
* The Autogen agent invokes one of the predefined tools:
  + **text\_to\_aql\_to\_text(query)** → Converts the query into an **AQL command** and executes it on ArangoDB.
  + **text\_to\_nx\_algorithm\_to\_text(query)** → Generates Python code for **NetworkX graph algorithms** and runs it.
  + **execute\_hybrid\_query(query)** → Runs **both AQL and NetworkX computations** for a more comprehensive answer.
  + **extract\_nx\_subgraph(query)** → Extracts a **relevant subgraph** based on query constraints.
  + **visualize\_metrics(query)** → Computes **graph-based metrics** and visualizes the results.

**4️⃣ Query Execution and Data Retrieval**

* The selected tool is executed **dynamically**.
* If the tool requires **AQL**, it runs a structured query on **ArangoDB** and fetches data.
* If it requires **graph analysis**, it runs a **NetworkX computation** (e.g., PageRank, Betweenness Centrality).
* If needed, the results are **post-processed** using GPT-4o.

**5️⃣ Response Generation**

* The AI agent **formats the response** in a **human-readable** way.
* If a visualization is required, it **generates a graph plot**.
* If a list of entities is needed, it **summarizes and ranks results** based on importance.

Example Output for:  
🟢 *"Identify the most influential symptom among patients with Diabetes using PageRank."*  
✅ Runs **AQL query** to fetch relevant symptoms.  
✅ Runs **PageRank on NetworkX** for symptom importance.  
✅ Returns **a ranked list of symptoms** with **explanations**.  
✅ Generates **a graph visualization** for easier interpretation.

**Two notebooks are provided**

1)Copy of Final ArangoDB.ipynb (For agentic ai with Langchain and Langraph)

2)Final ArangoDB(Autogen\_exp).ipynb(Autogen)

Queries to try

query\_graph("Who is the most popular person in the Graph? Explain why")

print(query\_graph("How strongly connected is the network? Used connected components."))

query\_graph("Translate this natural language query into AQL: 'Find all diseases treated by drug Aspirin'")

query\_graph("Identify the most influential symptom among patients diagnosed with Diabetes using PageRank analysis.")

print(query\_graph("List recommended treatments for the disease Hypertension"))

query\_graph("Identify the most influential drugs for disease id Asthma using PageRank analysis.")

query\_graph("Visualize only the Betweenness values for a subgraph of diseases and treatments.")

print(query\_graph("Check how drug type Ibuprofen interacts with other drugs.Use aql"))

print(query\_graph("How strongly connected is the network? Used connected components."))