ML for Process Industry Series

Knowledge Graphs & GraphRAG for Process Industry

A short introductory hands-on guide for process data scientists

2026

First Edition

Ankur Kumar

Dedicated to our spouses, families, friends, motherlands, and all the process data-science enthusiasts

अमृतं भुज्यते विद्ये भवतीमाश्रितैः परम्।

अन्ये तु बत दूयन्ते संसरन्त इतस्ततः॥

The wise who take refuge in knowledge relish the nectar of wisdom,
while others suffer endlessly in worldly cycles.

- *A popular Sanskrit shloka*

Knowledge Graphs & GraphRAG for Process Industry

www.MLforPSE.com

Copyright © 2026 Ankur Kumar

All rights reserved. No part of this book may be reproduced or transmitted in any form or in any manner without the prior written permission of the authors.
.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented and obtain permissions for usage of copyrighted materials. However, the authors make no warranties, expressed or implied, regarding errors or omissions, and assume no legal liability or responsibility for loss or damage resulting from the use of information contained in this book.

To request permissions, contact the authors at MLforPSE@gmail.com

First published: May 2026

About the Author

Ankur Kumar holds a PhD degree (2016) in Process Systems Engineering from the University of Texas at Austin and a bachelor's degree (2012) in Chemical Engineering from the Indian Institute of Technology Bombay. He currently works at Linde in the Advanced Digital Technologies & Systems Group in Linde's Center of Excellence, where he has developed several in-house machine learning-based monitoring and process control solutions for Linde's hydrogen and air-separation plants. Ankur's tools have won several awards both within and outside Linde. One of his tools, PlantWatch (a plantwide fault detection and diagnosis tool), received the 2021 Industry 4.0 Award by the Confederation of Industry of the Czech Republic. Ankur has authored or co-authored several peer-reviewed journal papers (in the areas of data-driven process modeling and monitoring), is a frequent reviewer for many top-ranked Journals, and has served as Session Chair at several international conferences. Ankur served as an Associate Editor of the Journal of Process Control from 2019 to 2021, and currently serves on the Editorial Advisory Board of Industrial & Engineering Chemistry Research Journal. Most recently, he was included in the 'Engineering Leaders Under 40, Class of 2023' by *Plant Engineering* Magazine.

Note to the readers

Jupyter notebooks and Python scripts with complete code implementations are available for download at https://github.com/ML-PSE/Knowledge_Graphs_and_GraphRAG_for_Process_Industry. Code updates, when necessary, will be made and updated on the GitHub repository. Updates to the book's text material will be available on Leanpub (www.leanpub.com) and MLforPSE website (https://mlforpse.com/books/). We would greatly appreciate any information about any corrections and/or typos in the book.

Series Introduction

In the 21st century, data science has become an integral part of the work culture at every manufacturing industry and process industry is no exception to this modern phenomenon. From predictive maintenance to process monitoring, fault diagnosis to advanced process control, machine learning-based solutions are being used to achieve higher process reliability and efficiency. However, few books are available that adequately cater to the needs of budding process data scientists. The scant available resources include: 1) generic data science books that fail to account for the specific characteristics and needs of process plants 2) process domain-specific books with rigorous and verbose treatment of underlying mathematical details that become too theoretical for industrial practitioners. Understandably, this leaves a lot to be desired. Books are sought that have process systems in the backdrop, stress application aspects, and provide a guided tour of ML techniques that have proven useful in process industry. This series '***Machine Learning for Process Industry***' addresses this gap to reduce the barrier-to-entry for those new to process data science.

The first book of the series '***Machine Learning in Python for Process Systems Engineering***' covers the basic foundations of machine learning and provides an overview of broad spectrum of ML methods primarily suited for static systems. Step-by-step guidance on building ML solutions for process monitoring, soft sensing, predictive maintenance, etc. are provided using real process datasets. Aspects relevant to process systems such as modeling correlated variables via PCA/PLS, handling outliers in noisy multidimensional dataset, controlling processes using reinforcement learning, etc. are covered.

The second book '***Machine Learning in Python for Dynamic Process Systems***' provides a guided tour along the wide range of available dynamic modeling choices. Emphasis is paid to both the classical methods (ARX, CVA, ARMAX, OE, etc.) and modern neural network methods. Applications on time series analysis, noise modeling, system identification, and process fault detection are illustrated with examples.

The third book of the series '***Machine Learning in Python for Process and Equipment Condition Monitoring, and Predictive Maintenance***' takes a deep dive into an important application area of ML, viz, prognostics and health management. ML methods that are widely employed for the different aspects of plant health management, namely, fault detection, fault isolation, fault diagnosis, and fault prognosis, are covered in detail. Emphasis is placed on conceptual understanding and practical implementations.

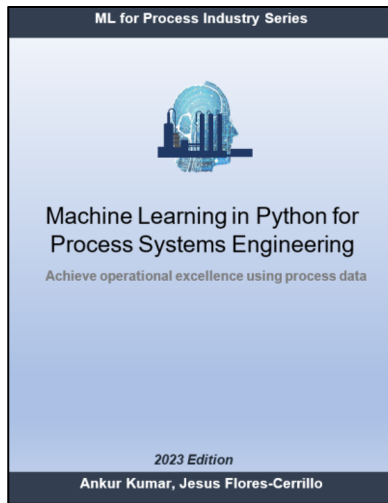
The fourth book of the series '***Machine Learning in Python for Visual and Acoustic Data-based Process Monitoring***' is a quick foray into the world of deep learning-based computer vision and abnormal equipment sound detection. The readers are introduced to the ease with which powerful equipment and product quality monitoring solutions can be built using sound and visual data.

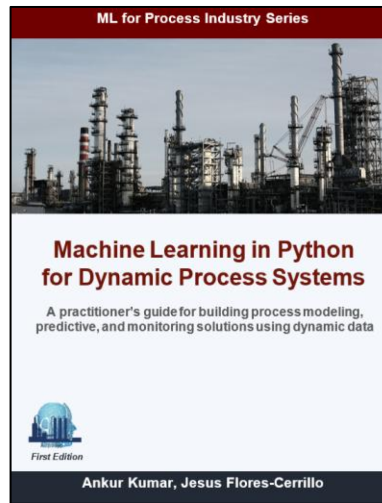
The fifth book of the series '***Building LLM and AI Agent-Based Applications for the Process Industry***' familiarizes readers with the world of LLM and agentic AI. It teaches readers how to build useful and reliable solutions for process industry operations and demonstrates the best practices for developing these applications. With no prerequisites required and a hands-on approach adopted throughout, this book makes advanced AI technologies accessible to process engineers and data scientists alike.

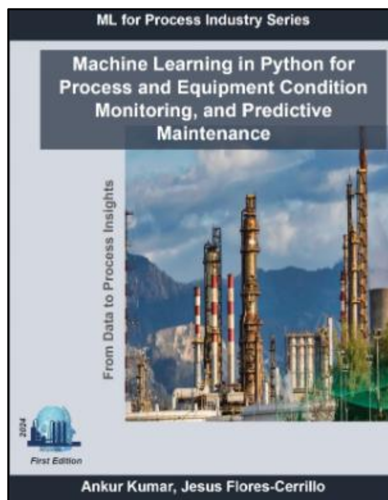
The current (sixth) book of the series ***'Knowledge Graphs and GraphRAG for Process Industry***' takes the LLM journey a step further. It introduces process data scientists to knowledge graphs and GraphRAG, a powerful combination that overcomes the limitations of vanilla retrieval-augmented generation when answering the kind of cross-document, relationship-heavy questions that process engineers usually care about. With a hands-on, application-driven (for plant troubleshooting) approach, the book walks readers through the complete pipeline: designing an ontology for plant data, using LLMs to extract entities and relationships from incident reports, building and querying the graph in Neo4j, and finally assembling everything into a demo web application.

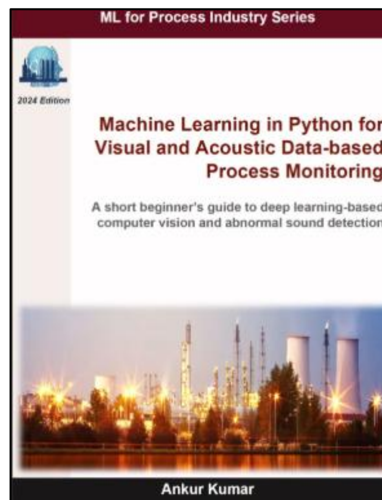
Future books of the series will continue to focus on other aspects and needs of the process industry. It is hoped that these books can help process data scientists find innovative ML solutions to the real-world problems faced by the process industry. With the growing trend in usage of machine learning in the process industry, there is growing demand for process domain experts/process engineers with data science/ML skills. These books have been written to cover the existing gap in ML resources for such process data scientists. Specifically, books of this series will be useful to budding process data scientists, practicing process engineers, process data science instructors, and ML/AI practitioners. All the books of the series are written keeping in mind the needs and characteristics of process systems. With the focus on practical guidelines and industrial-scale case studies, we hope that these books lead to wider spread of data science in the process industry.

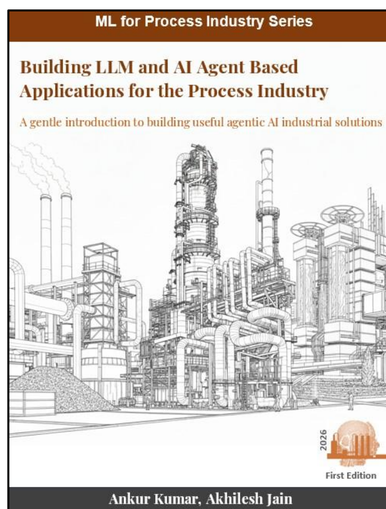
Other book(s) from the series
(**https://MLforPSE.com/books/**)











Preface

Imagine yourself in the shoes of a process engineer on a night shift. A critical pump in the crude unit is showing high vibration. The DCS confirms the suction pressure is dropping. You vaguely remember that something similar happened last year on a sister train, but you cannot recall the root cause or whether the corrective action actually held up. Somewhere in the plant's document management system, there is an incident report that has the answer. Somewhere in the maintenance log, there is a record of the technician who fixed it. Somewhere in the CMMS, there is a note about the upstream desalter that may have caused the problem in the first place. The knowledge exists. The connections exist. But there is no system that can pull them together for you in the next ten minutes. Most process plants today operate exactly this way, with priceless operational knowledge trapped inside thousands of unstructured documents that simply do not talk to each other.

If you have read the previous book of this series, you already know how Large Language Models and AI agents can serve as powerful reasoning engines, and how a simple retrieval-augmented generation (RAG) pipeline can ground their answers in your plant's documents. That approach works remarkably well for straightforward, single-document questions. But the most valuable questions in a process plant are rarely that simple. They are cross-document, multi-hop, and relationship-heavy: "What caused this last time? What downstream equipment is at risk? Who fixed it? Did the fix actually work?" These are the questions where simple RAG may not provide satisfactory answers, and they are precisely the questions that knowledge graphs and GraphRAG are designed to answer.

The process industry stands to benefit enormously from knowledge graphs: incident reports, maintenance work orders, operating procedures, P&IDs, and engineering change documents are all inherently graph-shaped, but the data has historically been locked away in text. Unfortunately, the resources available to a process engineer or data scientist who wants to build such applications are surprisingly inadequate. Missing from the landscape is a hands-on, application-oriented guide that places process industry operations at the center and walks the reader through the complete journey of designing an ontology, building the graph, querying it with natural language, and deploying the result as a usable application. This book attempts to fill that gap. It is not meant to be the ultimate reference on graph theory or graph databases; rather, it aims to help process data scientists and engineers take their first confident steps into the world of knowledge graphs and GraphRAG, understand the full picture end-to-end, and build a strong enough foundation to keep learning and building on their own.

What this short book offers

How can an AI assistant help a maintenance engineer diagnose a recurring compressor fault by pulling together information from a decade of incident reports? How can it trace a cascading failure from a desalter through a pump, a heat exchanger, and a reactor, across four different documents written by four different engineers? These are the questions that motivate this book.

We adopt a hands-on, tutorial-style approach throughout. Readers will learn how to design a domain ontology for plant data, set up a Neo4j graph database, write Cypher queries that traverse multi-hop relationships, use OpenAI's structured output capability to extract entities and relationships from unstructured incident reports, build a natural-language-to-Cypher query pipeline with LangChain, and combine graph traversal with vector search for hybrid retrieval. Every concept is accompanied by working code examples drawn from process industry operations, and the complete codebase (Jupyter notebooks, Python scripts, sample reports, and the final Streamlit application) is provided in the accompanying GitHub repository.

This short book is organized into five chapters that build on each other. Chapter 1 motivates the need for knowledge graphs and GraphRAG, explains where simple RAG falls short, and introduces the technology stack we will use. Chapter 2 takes you hands-on with Neo4j: setting up a free cloud database, learning Cypher fundamentals, and constructing a small troubleshooting graph by hand so that you develop an intuitive feel for how nodes, edges, and properties fit together. Chapter 3 automates that construction by using LLM-powered entity and relationship extraction to ingest incident reports into the graph at scale. Chapter 4 builds the query pipeline that turns a natural-language question into Cypher, executes it on the graph, and synthesizes a grounded answer; it also introduces hybrid retrieval that combines graph traversal with vector similarity. Chapter 5 brings everything together into a demo Streamlit web application where engineers can upload reports, watch the graph grow, ask questions, and inspect the generated Cypher for full transparency.

Who should read this book

The application-oriented approach in this book is meant to provide a comprehensive and practical guide to building knowledge graph and GraphRAG-based solutions for process industry operations. The following categories of readers will find the book useful:

1) Process engineers and operators who want to understand how knowledge graphs and GraphRAG-powered tools work, what they can realistically accomplish, and how to champion their adoption within their organizations

2) Data scientists and ML practitioners working in process industries who want to extend their RAG skill set with knowledge graphs and graph databases

3) Process data science instructors looking for a structured, example-driven resource to teach knowledge graphs and GraphRAG concepts in an industrial context

4) Industrial practitioners and technology leaders evaluating the potential of GraphRAG for their operations and seeking a grounded understanding of what it takes to build reliable solutions

Pre-requisites

Basic familiarity with Python and LLM API calls is assumed. Readers who have worked through the previous book of the series (or who have built a simple RAG system on their own) will find the transition smooth. No prior experience with graph databases or Cypher is required; the book introduces both from scratch. Complete code implementations are available in the accompanying GitHub repository.

Ankur Kumar

Table of Contents

Chapter 1: Why Knowledge Graphs and GraphRAG?

1.1 The Knowledge Problem in Process Plants
1.2 What is a Knowledge Graph?
1.3 Where Traditional Approaches Fall Short
1.4 GraphRAG: The Best of Both Worlds
1.5 The Pipeline We Will Build
1.6 Other Process Industry Use Cases

Chapter 2: Building Your First Knowledge Graph

2.1 Setting Up Neo4j AuraDB
2.2 Cypher: The Graph Query Language
2.3 Building the Mini Graph
2.4 Querying the Graph
2.5 Graph Visualization with pyvis

Chapter 3: LLM-Powered Entity Extraction

3.1 Designing the Graph Schema
3.2 Extracting Entities using LLM
3.3 Ingesting Extracted Entities into Neo4j Graph

Chapter 4: GraphRAG Query Pipeline

4.1 How GraphRAG Querying Works
4.2 GraphCypherQAChain with LangChain
4.3 Raw SDK Approach for Full Control
4.4 Template-Based Approach for Production Safety
4.5 Hybrid Retrieval: Graph + Vector Search

Chapter 5: Building a Complete GraphRAG Application

5.1 Application Architecture
5.2 The Streamlit App: Complete Code Walkthrough
-- The Upload → Extract → Ingest Flow
-- The Chat Interface with Cypher Transparency

Chapter 1

Why Knowledge Graphs and GraphRAG?

In the previous book of this book series, we learned how LLMs can serve as powerful reasoning engines, and how AI agents can query databases, search documents, and take actions on behalf of process engineers. We also built a simple RAG (Retrieval-Augmented Generation) system that retrieves relevant document chunks to ground the LLM's responses in factual plant data. That approach works remarkably well for straightforward questions, such as "What does the SOP say about starting up compressor C-401?"

But here's the thing: the most valuable questions in a process plant are rarely that simple. When a pump starts cavitating at 2 AM and the shift engineer needs answers, they're not asking for a single fact from a single document. They're asking questions that require connecting information scattered across dozens of reports, maintenance logs, and engineering records: "What caused this last time? What downstream equipment is at risk? Who fixed it? Did the fix actually work, or did the problem come back?"

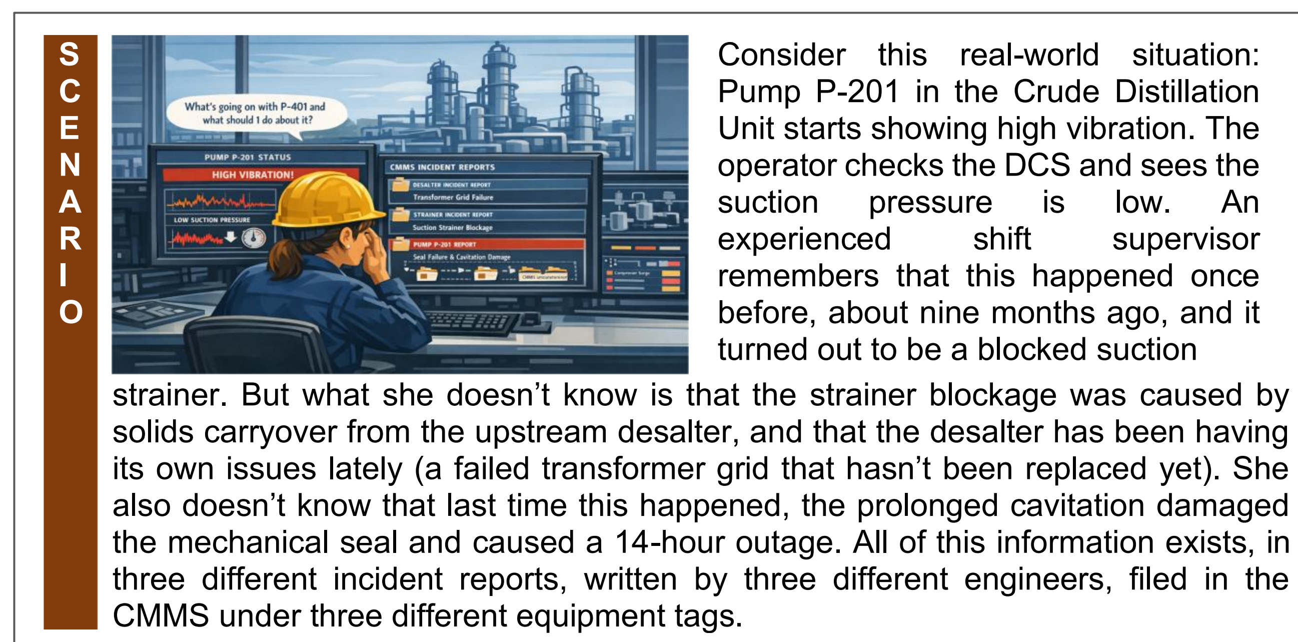
This chapter explains why those questions are hard for traditional approaches, and introduces knowledge graphs and GraphRAG as the solution. By the end, you'll understand the core concepts, see exactly where simple RAG breaks down, and have a clear picture of the pipeline we'll build together in the remaining chapters. Specifically, the following topics are covered

- Introduction to Knowledge Graphs and Graph RAG
- Shortcomings of traditional approaches (relational databases and simple RAG)
- Simple RAG vs GraphRAG
- Technology stack for process troubleshooting application

Let's now take our first step towards adding these potent techniques to our data analytics toolkit!

1.1 The Knowledge Problem in Process Plants

Every process plant generates enormous amounts of operational knowledge. Incident reports, troubleshooting logs, maintenance work orders, operating procedures, engineering change documents, the list goes on. Over years of operation, a typical refinery or chemical plant accumulates thousands of these documents, each containing valuable insights about how equipment behaves, fails, and gets fixed. The problem is that this knowledge is trapped; locked inside unstructured text documents, scattered across different systems (CMMS, document management, shared drives, even personal email), and organized in ways that make cross-referencing extremely difficult.



Consider this real-world situation: Pump P-201 in the Crude Distillation Unit starts showing high vibration. The operator checks the DCS and sees the suction pressure is low. An experienced shift supervisor remembers that this happened once before, about nine months ago, and it turned out to be a blocked suction strainer. But what she doesn't know is that the strainer blockage was caused by solids carryover from the upstream desalter, and that the desalter has been having its own issues lately (a failed transformer grid that hasn't been replaced yet). She also doesn't know that last time this happened, the prolonged cavitation damaged the mechanical seal and caused a 14-hour outage. All of this information exists, in three different incident reports, written by three different engineers, filed in the CMMS under three different equipment tags.

Illustration 1.1: Conceptual illustration of knowledge problem in process plants

The knowledge exists. The connections exist. But no system makes those connections visible. The experienced supervisor relies on her memory (fallible, incomplete, and walking out the door when she retires). The junior engineer has no way to discover these cross-document patterns at all. This is the knowledge problem that knowledge graphs solve.

1.2 What is a Knowledge Graph?

A knowledge graph is a way of organizing information as a network of entities and the relationships between them. Instead of storing data in rows and columns (like a spreadsheet) or as chunks of text (like a document database), a knowledge graph stores data as:

- **Nodes**: Things that exist: a piece of equipment (P-201), an incident (INC-2024-0012), a symptom (High vibration), a root cause (Blocked suction strainer), a corrective action (Replace mechanical seal), a person (Armaan).
- **Relationships**: How those things connect: the incident EXHIBITED a symptom, the incident was CAUSED_BY the root cause, it was RESOLVED_WITH the corrective action, and the action was PERFORMED_BY the technician.
- **Properties**: Additional details on nodes and relationships: the incident has a date (2024-01-18), a severity (High), and a downtime (14 hours). The CAUSED_BY relationship might have a confidence level (primary vs. contributing).

Figure 1.1 shows a simple example of what this looks like for one incident.

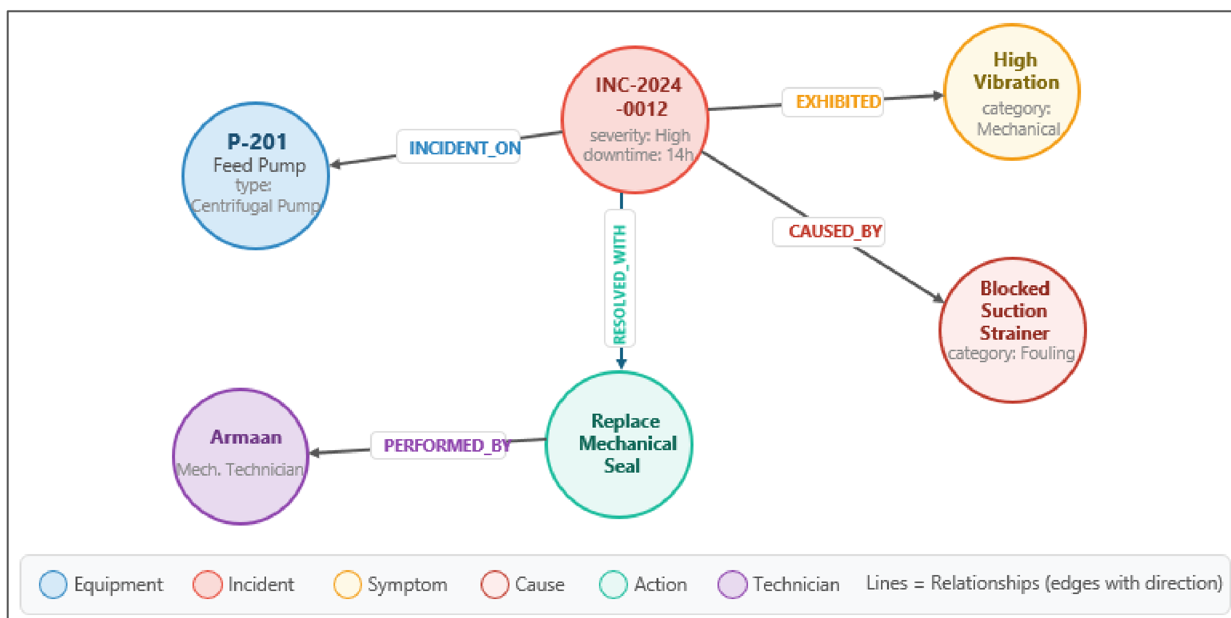


Figure 1.1: Anatomy of a Knowledge Graph

What makes this structure powerful is that you can follow the connections. Starting from INC-2024-0012, you can ask: “What equipment were impacted in this incident?” → “What were the symptoms?” → “What were the root causes?” → “What actions resolved them?” → “Who did the work?” Each question is answered by traversing one hop in the graph. And critically, these traversals work across all incidents; if P-201 appears in three different incident reports over two years, all three are connected to the same P-201 node.

Why Graphs Are Natural for Process Data

If you think about it, process plant data is inherently graph-shaped. Equipment connects to other equipment through piping and control signals. Incidents involve multiple pieces of equipment. Root causes cascade from one system to another. Technicians work across different equipment and units. These are all relationships, and they matter just as much as the entities themselves.

1.3 Where Traditional Approaches Fall Short

Relational Databases and JOIN Fatigue

If you're a process data scientist, your first instinct might be to put all this data in a relational database with tables for equipment, incidents, symptoms, causes, and actions. That's a perfectly reasonable starting point, and for simple lookups it works fine. But the moment you start asking relationship-heavy questions, the SQL gets painful. "What symptoms has P-201 shown across all incidents, and what were the root causes for each?" requires joining the equipment table to an incidents table, then to a junction table linking incidents to symptoms, then to another junction table linking incidents to causes. Four JOINs for a straightforward question.

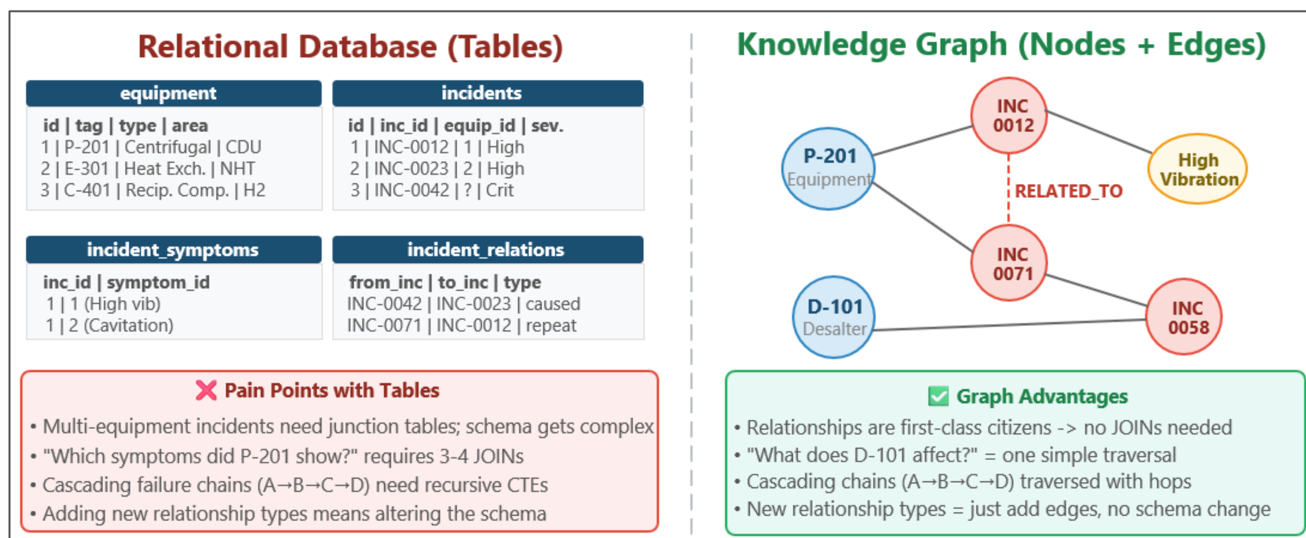
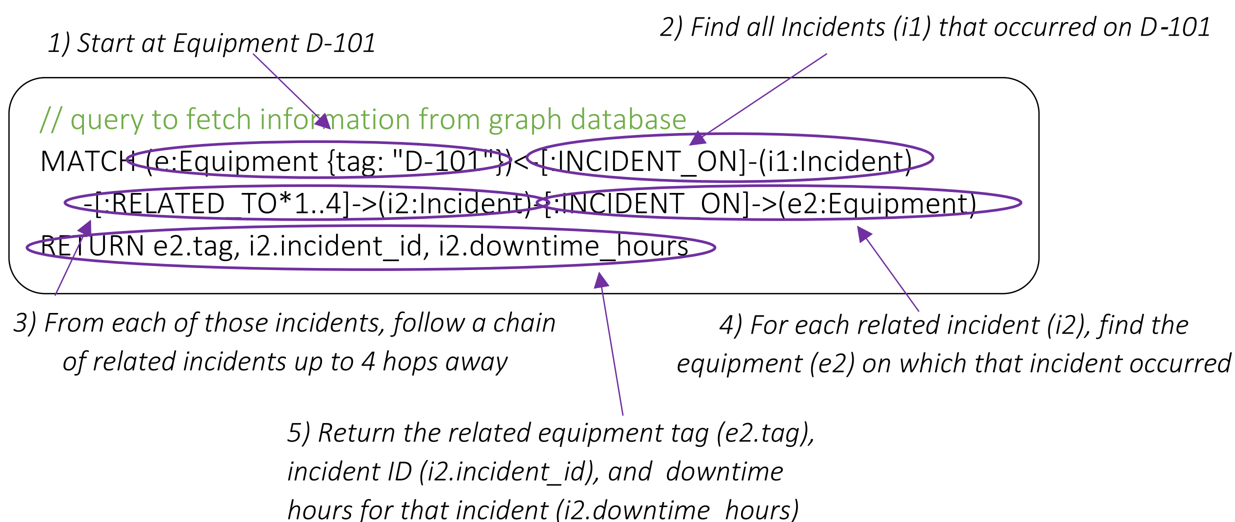


Figure 1.2: Relational Database vs Knowledge Graph for Process Data

Now try the cascading failure chain: "What downstream problems can desalter D-101 cause?" This requires recursive queries to traverse the chain D-101 → P-201 → E-301 → R-301. The SQL is verbose, hard to read, and error-prone. In a graph database, the same query is



One query (translated to "*Find all incidents connected (within 4 hops) to incidents on D-101, and list the equipment and downtime associated with those related incidents.*"). No JOINs. The graph database is doing exactly what the graph structure was designed for: **following paths**!

There's another practical issue: schema rigidity. When you discover a new kind of relationship (say, CONNECTED_TO between equipment for upstream/downstream topology), adding it to a relational database means altering the schema, creating a new junction table, and updating all the application code that touches it. In a graph, you just add edges. The schema is flexible by design.

Simple RAG and the Chunk Boundary Problem

In traditional RAG, we split documents into chunks, embedded them as vectors, and retrieved the most similar chunks for a given question. This works well when the answer lives inside one or two chunks. But process troubleshooting questions often require information from multiple documents that share no textual similarity; they are connected by relationships, not by keywords.

Consider this question: *"Our desalter D-101 is having issues. What downstream problems should we watch out for?"* With simple RAG, the vector search retrieves the desalter incident report (because it's the most similar to the question). The LLM reads that report and summarizes the desalter problem. It may retrieve another report on P-201 cavitation caused by D-101 issues; however, it will completely miss that P-201 cavitation is linked to E-301 fouling (another report) and it contributed to an R-301 temperature runaway (yet another report). The later reports share no common keywords: they are connected by causal relationships that only exist in the graph.

This is the fundamental limitation: simple RAG retrieves by textual similarity, but the answer requires relational reasoning across documents. No amount of chunking strategy or embedding model improvement can fix this; it's an architectural limitation!

1.4 Enter GraphRAG: The Best of Both Worlds

GraphRAG solves the aforementioned problem by adding a knowledge graph layer between your documents and your LLM. Instead of searching for similar text chunks, GraphRAG converts the user's natural language question into a graph query, executes it against the knowledge graph, and passes the structured results to the LLM for answer generation. Figure 1.3 below shows how GraphRAG works; the pipeline has three phases:

- **Build the graph**: Extract entities and relationships from your documents using an LLM, and store them in a graph database (Neo4j[1]).

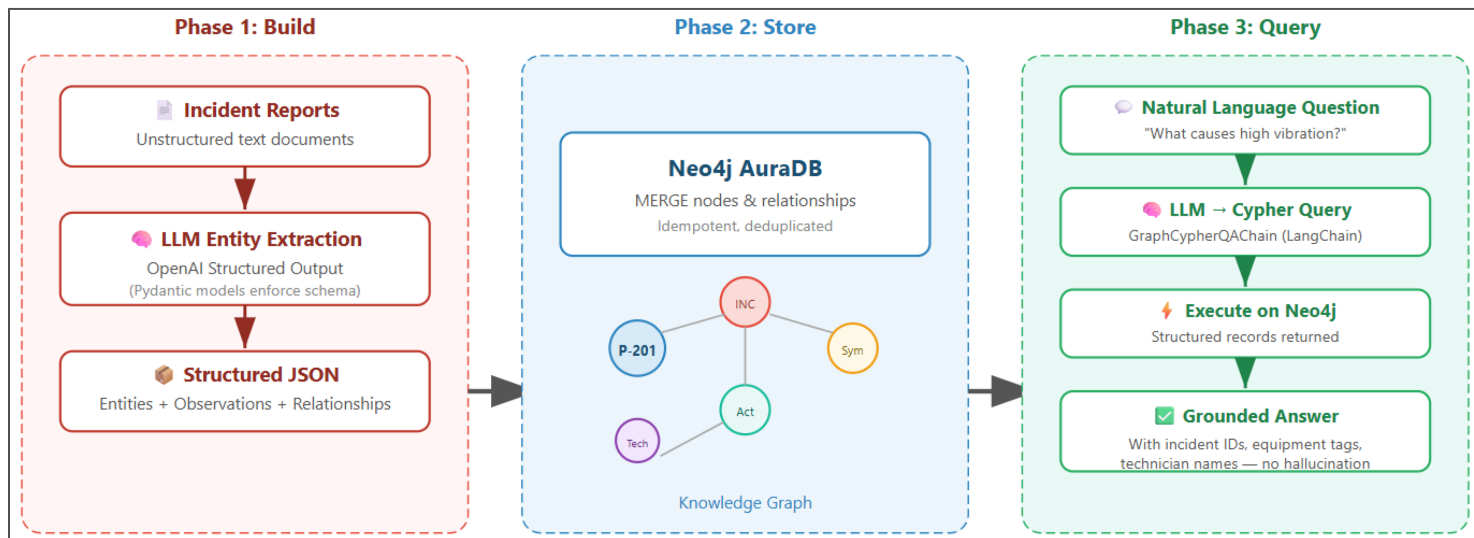


Figure 1.3: The GraphRAG Pipeline: From Documents to Answers

- **Store the graph**: The graph persists in the database, accumulating knowledge as more documents are ingested. Unlike vector stores, the graph captures the structure of the knowledge, not just the content.

[1] https://neo4j.com/

➢ **Query the graph**: When a user asks a question, an LLM generates a Cypher query (the graph query language), executes it on Neo4j, and another LLM call synthesizes the answer from the structured results.

Simple RAG vs GraphRAG: Head to Head

Let's make the GraphRAG pipeline concrete with the desalter question that we previously alluded to:

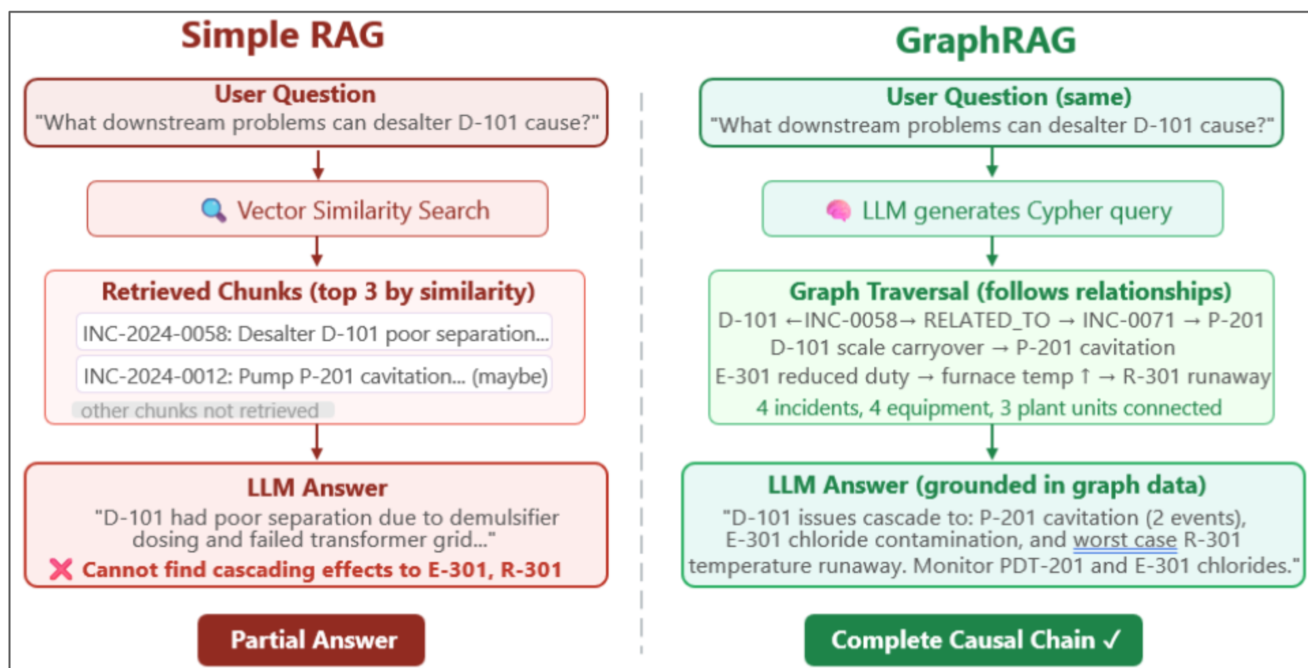


Figure 1.4: Simple RAG vs GraphRAG: How They Answer Differently]

The difference is dramatic. Simple RAG gives you one incident's information. GraphRAG gives you a complete causal chain across four incidents and four pieces of equipment, with specific corrective actions and monitoring recommendations for each. This is not an incremental improvement, but a qualitatively different kind of answer.

When GraphRAG Wins (and When It Doesn't)

GraphRAG is not always better than simple RAG. For straightforward factual lookups ("What does the SOP say about compressor startup?"), simple RAG works just fine. GraphRAG adds value when the question requires:

- **Multi-hop reasoning:** “What downstream problems can D-101 cause?” → requires traversing causal chains across multiple incidents.

- **Cross-document aggregation:** “What are all causes of high vibration in our plant?” → requires scanning every incident that exhibited this symptom.

- **Expert routing:** “Who should I call for compressor problems?” → requires traversing from equipment through incidents to technicians, ranked by experience.

- **Pattern detection:** “Which equipment has repeat failures?” → requires grouping incidents by equipment and comparing root causes.

- **Multi-equipment disambiguation:** “In the pump-motor incident, which equipment showed which symptom?”

- **Quantitative aggregation:** “Which root cause category caused the most downtime?” → requires summing across all incidents.

If your application primarily handles simple factual lookups, simple RAG is sufficient (and simpler to build). But if your users ask the kinds of cross-cutting, relationship-heavy questions that process engineers actually care about, GraphRAG is the right tool.

1.5 The Pipeline We Will Build

Throughout this book, we will build a complete GraphRAG pipeline for process troubleshooting, step by step. Here is the technology stack and what each chapter covers:

Chapter	Focus	What You’ll Build
1	Concepts & motivation	Understanding of KGs, GraphRAG, and why they matter
2	Neo4j Graph Database & Cypher	A hand-built troubleshooting graph with Cypher queries
3	LLM-based extraction	Automated entity extraction from 10 incident reports using OpenAI API calls, ingested into Neo4j
4	Query pipeline	GraphCypherQAChain (langChain) based query pipeline; plus hybrid graph+vector retrieval
5	Streamlit app	A complete web application: upload reports, build graph, ask questions, see generated Cypher

The Troubleshooting Theme

Our running example throughout the book is a set of 10 realistic synthetic incident reports[2] from a petroleum refinery. These reports cover equipment failures across the Crude Distillation Unit, Naphtha Hydrotreater, Hydrogen Recovery Unit, and Utilities. One of the reports is shown below.

```
INC-2024-0012.txt ×
reports > INC-2024-0012.txt
 1  INCIDENT REPORT: INC-2024-0012
 2  Date: 2024-01-18
 3  Unit: Crude Distillation Unit (CDU)
 4  Equipment: Centrifugal Pump P-201 (Feed Pump)
 5  Severity: High | Downtime: 14 hours
 6
 7  OPERATING CONTEXT:
 8  Plant was running at 92% capacity on Arabian Light crude. Suction strainer S-201 cleaning interval had been
    extended from 30 to 90 days six months prior as a cost-saving measure.
 9
10  OBSERVED SYMPTOMS:
11  - P-201: Cavitation noise (rattling from impeller area) detected by operator Rajesh Patel at 02:15
12  - P-201: High vibration on bearing housing — measured 12.5 mm/s (alarm threshold 8.0 mm/s)
13  - P-201: Erratic flow readings on FT-201
14  - P-201: Seal leak — visible hydrocarbon leak at mechanical seal face
15
16  INVESTIGATION:
17  Mechanical technician Ahmed Khan inspected the pump. Suction strainer S-201 was 70% blocked with scale
    deposits from crude desalter D-101 upstream. Low NPSH caused vapor bubble formation in the impeller eye.
    Prolonged cavitation had damaged the mechanical seal.
18
19  ROOT CAUSE:
20  Primary: Blocked suction strainer S-201 due to upstream scale carryover from desalter D-101.
21  Contributing: Extended strainer cleaning interval (30 to 90 days).
22
23  CORRECTIVE ACTIONS:
24  1. Isolated and drained pump P-201. Replaced mechanical seal (4 hours). [Ahmed Khan]
25  2. Cleaned suction strainer S-201 and installed differential pressure transmitter PDT-201 for continuous
    monitoring. [Priya Sharma]
26  3. Restored strainer cleaning schedule to 30-day intervals. [David Kim]
27  4. Recommendation: inspect desalter D-101 wash water injection nozzles for scaling.
28
29  OUTCOME:
30  P-201 restored to service. PDT-201 providing early warning capability. No recurrence expected at 30-day
    cleaning interval.
31
32  TECHNICIANS: Ahmed Khan (Mechanical Technician), Priya Sharma (Instrument Technician), David Kim
    (Maintenance Planner)
```

The reports have been deliberately designed with interconnected patterns which would invisible to simple RAG but trivially discoverable in a knowledge graph:

- **Repeat failures:** Pump P-201 cavitates twice (INC-0012 and INC-0071). Compressor C-401 has two valve plate failures (INC-0031 and INC-0078).

[2] Provided in the GitHub repository

- **Cascading failures:** Desalter D-101 issues cascade to pump P-201 cavitation, which connects to heat exchanger E-301 fouling, which contributes to reactor R-301 temperature runaway.

- **Multi-equipment incidents:** A coupled pump-motor train (P-301/M-301) fails with different symptoms on each machine. Compressor C-401 and knockout drum V-401 have interlinked failures.

Other Technology Options for Building Graph-based Solution

Building a GraphRAG system requires choosing tools for three distinct jobs: storing the graph, extracting entities from text, and querying the graph with natural language. You would be lucky if you find a single package / platform that handles all the steps satisfactorily. The table below compares some popular options that you have for building your graphs

Tool	What it does	Strengths	Limitations
Neo4j Graph database + Cypher	Production-grade graph database. Cypher query language. Free managed cloud tier (AuraDB).	✓ Best LangChain integration ✓ Scales to production ✓ Rich ecosystem	Needs cloud account or Docker for self-hosted deployment
NetworkX Python graph library	In-memory graph data structure. Pure Python, zero setup. Great for prototyping.	✓ No installation ✓ Rich graph algorithms ✓ Best for testing ontology designs	✗ No persistence ✗ No query language ✗ Doesn't scale well
LlamaIndex PropertyGraph-Index	All-in-one framework: extracts entities, builds graph, queries: all in ~20 lines of code.	✓ Fast prototype ✓ Handles chunking and embeddings ✓ Good documentation	✗ Generic entity types (Person, Event) ✗ Cannot enforce your Domain-specific ontology
Microsoft GraphRAG	Document-focused framework. Community detection + hierarchical summarization.	✓ Strong for long narrative text ✓ Multi-level summarization	✗ Not for controlled industrial schemas ✗ Better as document layer, not plant graph
Cognee LLM memory engine	Auto-extracts entities from documents. Flexible graph backend,	✓ Fast experimentation ✓ Built-in entity resolution	✗ Less schema control

Our process troubleshooting application requires a carefully designed domain ontology: equipment nodes with tag numbers, observation junction nodes for multi-equipment disambiguation, canonicalized symptoms, etc. Our stack (*Neo4j + OpenAI Structured Output + LangChain*) gives full control over the schema while keeping each component simple and independently replaceable.

1.6 Other Process Industry Use Cases

While we focus on process troubleshooting as the demo GraphRAG application in this book, knowledge graphs have broader applications across the process industry. Below are a few other common applications:

- **P&ID as Knowledge Graph[3]:** Convert piping and instrumentation diagrams into a graph of equipment, instruments, and piping connections. Augment with live sensor data. Ask: "*What is upstream of heat exchanger E-301*?" or "*Show all instruments in alarm in the distillation section*."

- **Predictive Maintenance[4]:** Model machines, components, operating conditions, asset performance data, and failure histories as a graph. Query: "*Which pumps with similar operating profiles have experienced seal failures within 6 months*?"

- **Supply Chain & Recipe Optimization[5]:** Connect products, recipes, raw materials, and suppliers. Query: "If Supplier X can't deliver catalyst Y, which alternative recipes can produce Product Z meeting customer spec?"

- **Knowledge Fusion Across Silos:** Link a "pump" mentioned in a maintenance work order to the same pump's tag in the DCS, its spec sheet in the document management system, and its purchase record in SAP.

Knowledge Graphs in the Wild

Knowledge graphs are not new or experimental. Google's Knowledge Graph powers the information panels you see in search results. LinkedIn uses a knowledge graph with billions of nodes to power "People You May Know." Amazon uses a product knowledge graph for recommendations. What is new is the combination of knowledge graphs with LLMs: using LLMs to build the graph (extraction) and using the graph to ground the LLM's responses (retrieval). This combination is what we call GraphRAG.

[3] https://devblogs.microsoft.com/ise/engineering-document-pid-digitization/
https://arxiv.org/abs/2603.22528
[4] https://ceur-ws.org/Vol-2180/paper-86.pdf
[5] https://blog.siemens.com/en/2025/08/next-generation-customer-understanding-a-knowledge-graph-powered-ai-solution/

Summary

This chapter motivated why knowledge graphs and GraphRAG matter for process industry applications. We saw that traditional approaches, viz, relational databases and simple RAG, struggle with the relationship-heavy, cross-document questions that process engineers actually need to answer. Knowledge graphs organize information as networks of entities and relationships, making those cross-cutting queries natural and efficient. GraphRAG combines this graph structure with LLM-powered querying to deliver grounded, multi-hop answers that simple RAG cannot provide.

In the next chapter, we will get our hands dirty. You will set up a Neo4j graph database, learn the Cypher query language through process industry examples, and build your first troubleshooting knowledge graph by hand, experiencing firsthand the power of graph-based data organization before we automate it with LLMs in Chapter 3.

Chapter 2

Building Your First Knowledge Graph

In Chapter 1, we established why knowledge graphs and GraphRAG matter. We saw that traditional approaches (relational databases and simple RAG) struggle with the cross-document, relationship-heavy questions that process engineers actually need to answer. Now it's time to build something real.

In this chapter, you will set up a Neo4j graph database, learn the Cypher query language through process industry examples, and construct a mini troubleshooting knowledge graph entirely by hand. By the end, you'll be able to create nodes and relationships, query multi-hop paths, detect cascading failure chains, and what deserves to be a first-class entity (a node) versus what should be stored as a property. This hands-on experience will prepare you for Chapter 3, where we automate the entire process using LLMs.

Specifically, the following topics are covered

- Setting up Neo4j AuraDB and connecting from Python
- Cypher query language fundamentals: CREATE, MERGE, MATCH,
- Building a mini troubleshooting graph by hand
- Edge properties
- Querying the graph
- Graph visualization with pyvis

2.1 Setting Up Neo4j AuraDB

Neo4j is among the most popular graph database. It stores data as nodes and relationships (as we discussed in Chapter 1) and provides Cypher, a purpose-built query language for traversing graph patterns. For this book, we'll use Neo4j AuraDB, a free and fully-managed cloud instance that requires no installation: you sign up, get a database, and connect from Python in under three minutes.

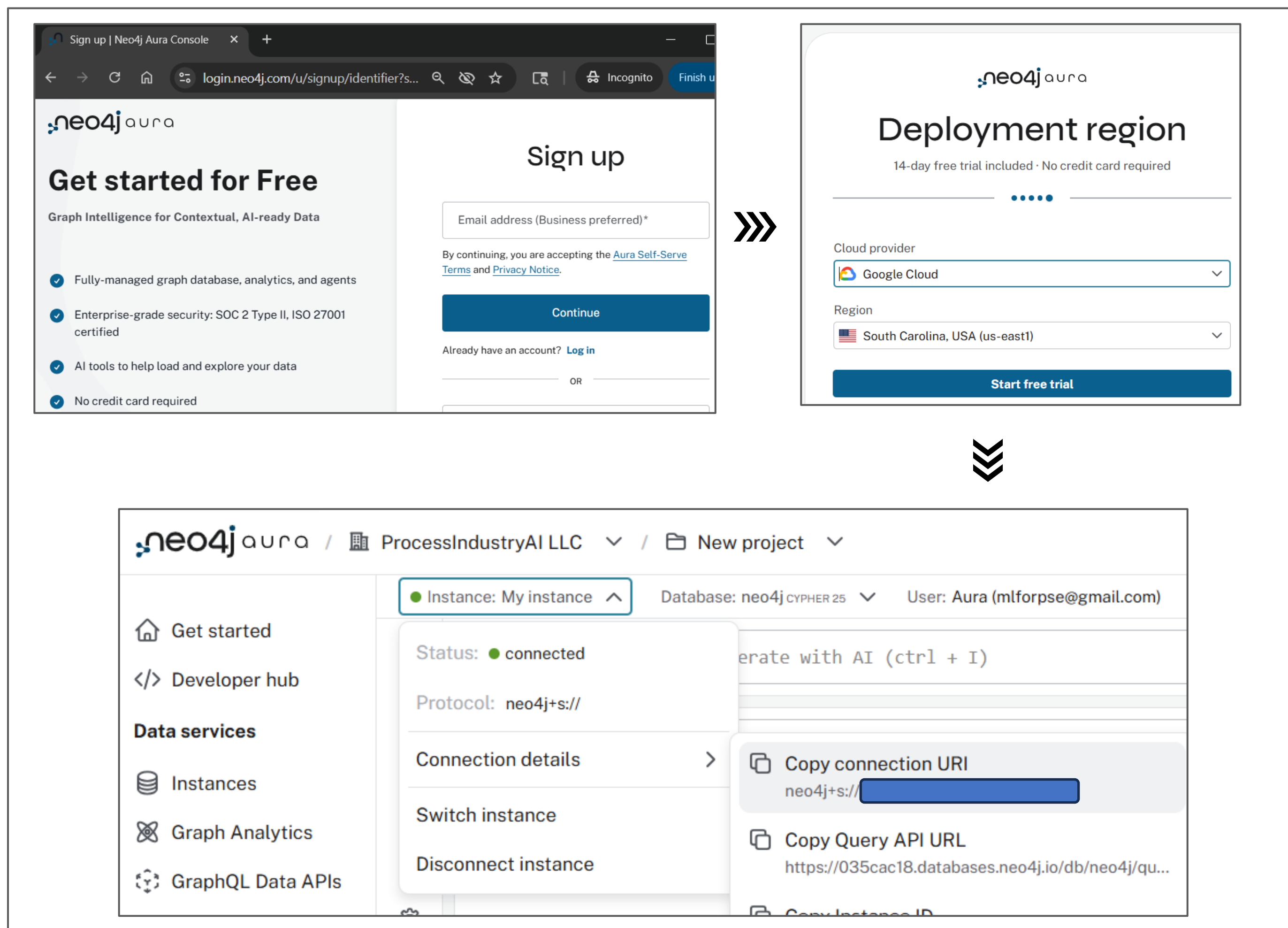


Figure 2.1: Setting up Neo4j Graph Database

Creating your free instance (Figure 2.1)

- Go to https://neo4j.com/ and sign up
- Start a free trial and save the auto-generated password
- Note the connection URI as shown in the figure above
- Create a .env file in your project directory with your credentials.

```
# .env file
NEO4J_URI=neo4j+s://xxxxxxxx.databases.neo4j.io
NEO4J_USERNAME=neo4j
NEO4J_PASSWORD=your-auto-generated-password
```

Connecting from Python

With your credentials saved in a .env file, connecting from Python is straightforward:

```
# imports
from neo4j import GraphDatabase
from dotenv import load_dotenv
load_dotenv()

 # connect to Neo4j
driver = GraphDatabase.driver(
    os.getenv('NEO4J_URI'),
    auth=(os.getenv('NEO4J_USERNAME'), os.getenv('NEO4J_PASSWORD')))

# quick test
with driver.session() as session:
    result = session.run('RETURN "Hello from Neo4j!" AS message')
    print(result.single()['message'])

>>> Hello from Neo4j!
```

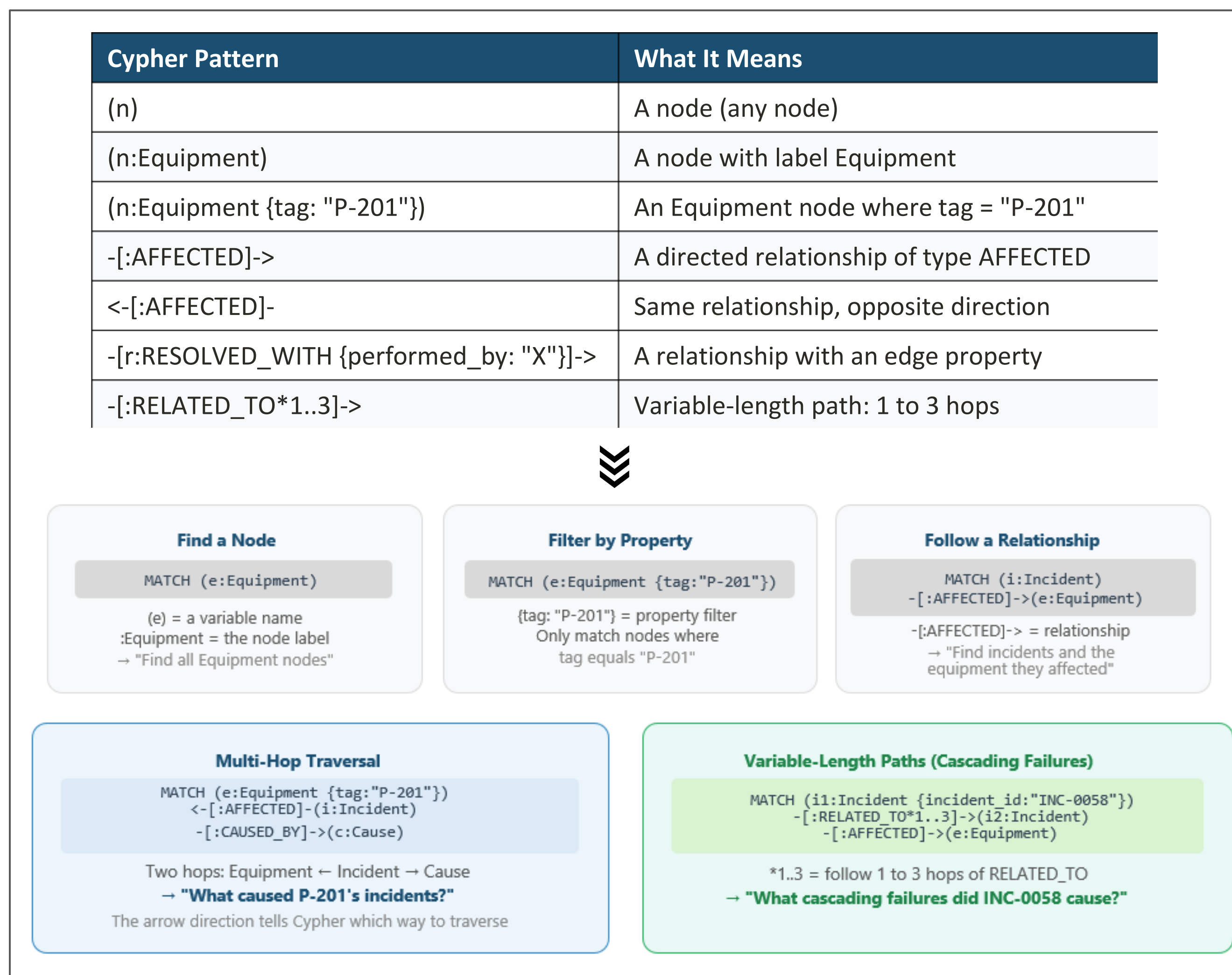
The accompanying Jupyter notebook (*ch2_building_your_first_kg.ipynb*) contains all the code that you will see in this chapter. If you see the above message after executing your Jupyter Notebook cell, you're ready to start building your graph.

Neo4j AuraDB vs Docker

For learning and prototyping, AuraDB Free is the pragmatic choice. An alternative is running Neo4j in Docker locally. All the Cypher code in this book works identically on both.

2.2 Cypher: The Graph Query Language

Cypher is to graph databases what SQL is to relational databases, but optimized for expressing graph patterns. If you know SQL, Cypher may feel familiar but simpler. The key insight is that Cypher uses ASCII art to represent graph patterns. This makes them remarkably intuitive for relationship-heavy queries; queries visually resemble the graph patterns they describe. Below we show some example patterns and a few sample usages of these patterns.



Cypher Pattern	What It Means
(n)	A node (any node)
(n:Equipment)	A node with label Equipment
(n:Equipment {tag: "P-201"})	An Equipment node where tag = "P-201"
-[:AFFECTED]->	A directed relationship of type AFFECTED
<-[:AFFECTED]-	Same relationship, opposite direction
-[r:RESOLVED_WITH {performed_by: "X"}]->	A relationship with an edge property
-[:RELATED_TO*1..3]->	Variable-length path: 1 to 3 hops

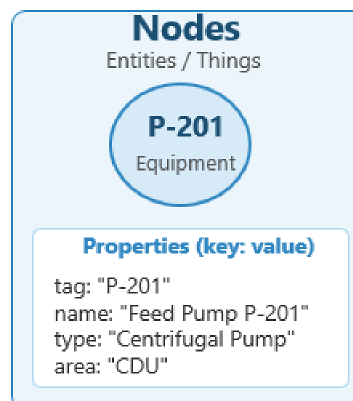
Figure 2.2: Reading Cypher - The Pattern-Matching Language

Let us work on some basic graph operation by hand.

Creating Nodes

CREATE makes a new node[6]. The label (after the colon) defines the node type, and properties are key-value pairs inside curly braces:

```
# Create our first node: Equipment P-201
CREATE (e:Equipment {
      tag: "P-201",
      name: "Feed Pump P-201",
      type: "Centrifugal Pump",
      area: "Crude Distillation Unit"})
```



To find the node we just created, we use the MATCH command:

```
MATCH (e:Equipment {tag: "P-201"})
RETURN e.tag, e.name, e.type, e.area
```

>>>

	tag	name	type	area
0	P-201	Feed Pump P-201	Centrifugal Pump	CDU

CREATE vs MERGE

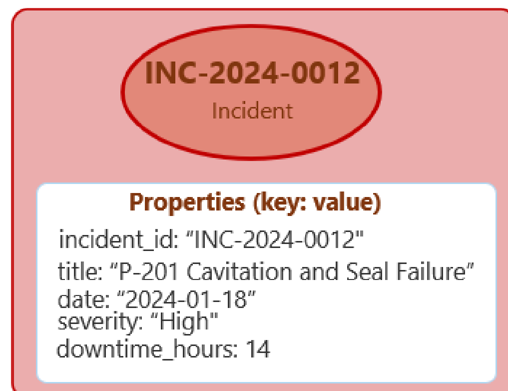
CREATE always makes a new node, even if an identical one already exists. This means running the same CREATE statement twice produces duplicate nodes, a serious problem when the same equipment appears in multiple incident reports. MERGE solves this: if a node with tag 'P-201' already exists, MERGE reuses it and updates its properties with SET. If it doesn't exist, MERGE creates it. This is idempotent, i.e., run it once or a hundred times, you always get exactly one P-201 node. Throughout this book, we use MERGE exclusively for node and relationship creation.

[6] The Jupyter notebook provides a couple of helper function that runs any Cypher query and returns the results as a pandas DataFrame

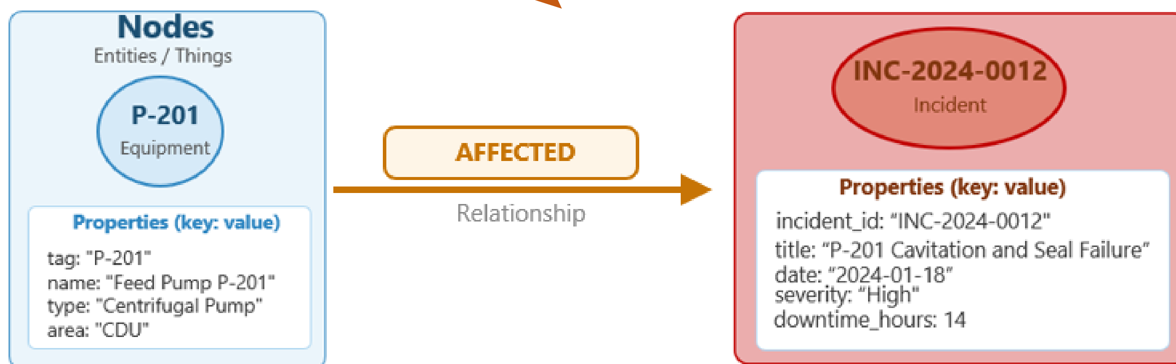
Creating Relationships

After creating nodes, we connect them with relationships. The pattern reads like a sentence; check the example below

```
# Create another node: an Incident node
MERGE (i:Incident {
    incident_id: "INC-2024-0012",
    title: "P-201 Cavitation and Seal Failure",
    date: "2024-01-18",
    severity: "High",
    downtime_hours: 14})
```




```
# Connect incident to equipment
MATCH (i:Incident {incident_id: "INC-2024-0012"})
MATCH (e:Equipment {tag: "P-201"})
MERGE (i)-[:AFFECTED]->(e)
```



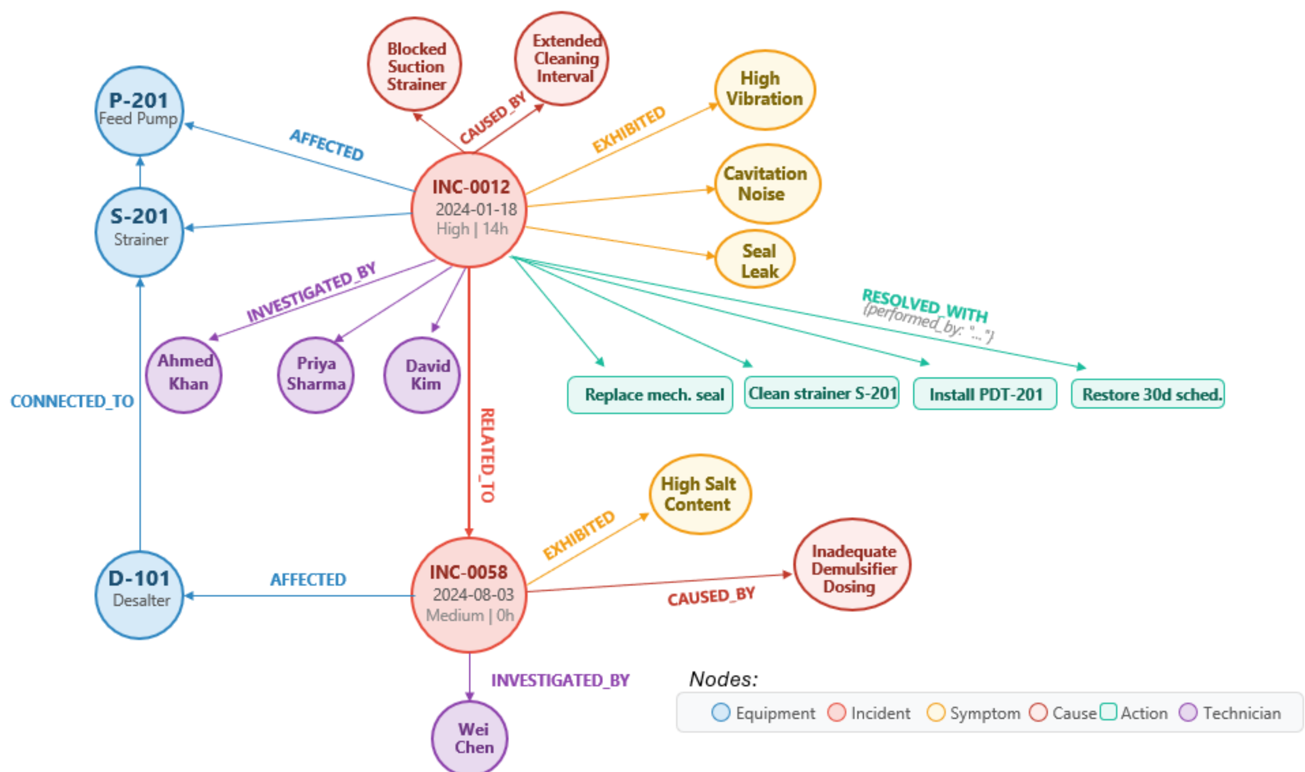
Core Concepts

1. Nodes have a label (:Equipment, :Incident) that defines their type, plus key-value properties
2. Relationships have a type (AFFECTED), are always directed (→), and can also carry properties

The above command is read as: "Find the incident INC-2024-0012 and the equipment P-201, then create an AFFECTED relationship between them." The arrow -> indicates direction; the incident affected the equipment, not the other way around.

2.3 Building the Mini Graph

To understand the process of creating a graph, you will now build a mini graph using a couple of incident reports as shown below. We'll create two incidents: the P-201 cavitation event (INC-2024-0012; we saw this incident's report in Chapter 1) and the D-101 desalter problem (INC-2024-0058), with a RELATED_TO link between them showing the cascading failure pattern.



The notebook walks through this construction step by step creating 3 equipment nodes, 2 incidents, symptoms, causes, and actions. Each step uses MERGE with verification queries so you can see the graph grow. You are encouraged to see the commands executed in the Notebook, but here is the overall flow:

- *Create equipment nodes: P-201, S-201 (suction strainer), D-101 (upstream desalter).*
- *Create the first incident node (INC-2024-0012) with its properties.*
- *Create symptom, cause, action, and technician nodes.*
- *Wire everything together with relationships: AFFECTED, EXHIBITED, CAUSED_BY, RESOLVED_WITH, INVESTIGATED_BY.*
- *Repeat for the second incident (INC-2024-0058).*
- *Connect the two incidents.*

An important design decision in the above mini graph was to record who performed each action in the *performed_by* as a property on the *RESOLVED_WITH* edge[7]:

```
MERGE (a:Action {name: "Replace mechanical seal"})
SET a.category = "Replacement"
WITH a
MATCH (i:Incident {incident_id: "INC-2024-0012"})
MERGE (i)-[r:RESOLVED_WITH]->(a)
SET r.performed_by = "Ahmed Khan"
```

The query for “who performed the seal replacement in INC-0012?” reads the edge property:

```
MATCH (i:Incident {incident_id: "INC-2024-0012"})
    -[r:RESOLVED_WITH]->(a:Action)
RETURN a.name AS Action, r.performed_by AS Performed_By
```

2.4 Querying the Graph

With our mini graph built, let’s ask a few illustrative questions.

Equipment History

```
MATCH (e:Equipment {tag: "P-201"})<-[:AFFECTED]-(i:Incident)
RETURN i.incident_id AS Incident, i.date AS Date,
       i.severity AS Severity, i.downtime_hours AS Downtime_Hrs
ORDER BY i.date

>>>
```

	Incident	Date	Severity	Downtime_Hrs
0	INC-2024-0012	2024-01-18	High	14

One hop: Equipment ← Incident. Returns all incidents that affected P-201.

[7] 'WITH a' acts as a pipeline separator in Cypher. It says "take the result from everything above and pass it forward to everything below."

Root Cause Analysis

```
MATCH (e:Equipment {tag: "P-201"})
      <-[:AFFECTED]-(i:Incident)
      -[:CAUSED_BY]->(c:Cause)
  RETURN i.incident_id AS Incident, c.name AS Root_Cause, c.category AS Category
```

```
>>>
```

	Incident	Root_Cause	Category
0	INC-2024-0012	Blocked suction strainer due to upstream scale...	Fouling
1	INC-2024-0012	Extended strainer cleaning interval	Human Error

Two hops: Equipment ← Incident → Cause. Returns root causes for all P-201 incidents.

Cascading Failure Chain

```
MATCH (e:Equipment {tag: "D-101"})<-[:AFFECTED]-(i1:Incident)
OPTIONAL MATCH (i1)<-[:RELATED_TO*1..3]-(i2:Incident)-[:AFFECTED]->(e2:Equipment)
RETURN i1.incident_id AS Origin_Incident, e.tag AS Origin_Equipment,
     i2.incident_id AS Cascaded_Incident, e2.tag AS Affected_Equipment
```

```
>>>
```

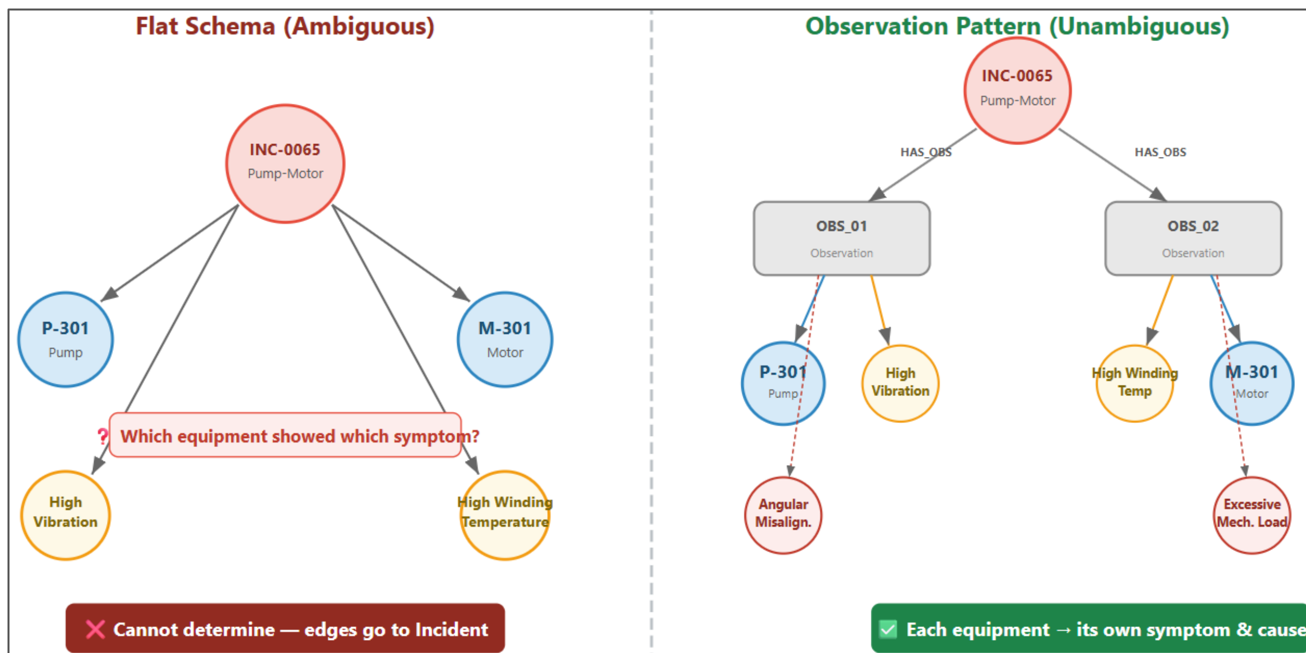
	Origin_Incident	Origin_Equipment	Cascaded_Incident	Affected_Equipment
0	INC-2024-0058	D-101	INC-2024-0012	P-201
1	INC-2024-0058	D-101	INC-2024-0012	S-201

This query follows RELATED_TO edges up to 3 hops from D-101's incident to find all downstream incidents in the causal chain. In our mini graph, it discovers that D-101 issues (INC-0058) caused P-201 cavitation (INC-0012). With a full 10-incident graph (Chapter 3), the chain extends further: D-101 → P-201 → E-301 → R-301.

2.5 The Observation Pattern

Everything so far has used a flat schema: Incident → EXHIBITED → Symptom, Incident → AFFECTED → Equipment. This works perfectly when each incident involves one piece of

equipment. But process plants often have coupled failures. Consider a coupled pump-motor failure where pump P-301 shows high vibration (caused by angular misalignment) and motor M-301 shows high winding temperature (caused by excessive mechanical load from the same misalignment). In the flat schema, both symptoms connect to the Incident and there is no way to determine which equipment showed which symptom as shown below.



The Solution: The Observation Node

We introduce a junction node called Observation that sits between the Incident and the Equipment/Symptom pair as shown in the figure above:

```
Incident — HAS_OBSERVATION —→ Observation — ON_EQUIPMENT —→ P-301
                                           — OBSERVED_SYMPTOM —→ High vibration
                                           — LOCAL_CAUSED_BY —→ Angular misalignment
```

Each Observation bundles ONE equipment with ONE symptom, plus optional local causes and actions. The query for “which equipment showed which symptom?” is now unambiguous:

```
MATCH (i:Incident {incident_id: "INC-2024-0065"})
    -[:HAS_OBSERVATION]->(ob:Observation)
    -[:ON_EQUIPMENT]->(e:Equipment)
MATCH (ob)-[:OBSERVED_SYMPTOM]->(s:Symptom)
RETURN e.tag AS Equipment, s.name AS Symptom
```

```
>>>
```

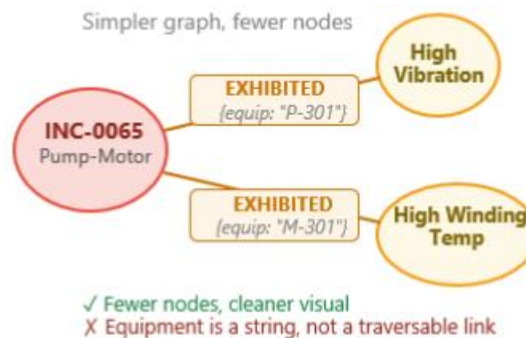
	Equipment	Symptom
0	P-301	High vibration
1	M-301	High winding temperature

Not every cause or action belongs to one equipment. Some are shared across the whole incident. The Observation pattern supports both levels:

- **Observation-level**: *LOCAL_CAUSED_BY* and *LOCAL_RESOLVED_BY* for equipment-specific causes and actions.
- **Incident-level**: *CAUSED_BY* and *RESOLVED_BY* for shared causes and actions (e.g., "Added post-restart alignment verification to procedure").

An Alternative: Edge Properties Instead of Junction Nodes

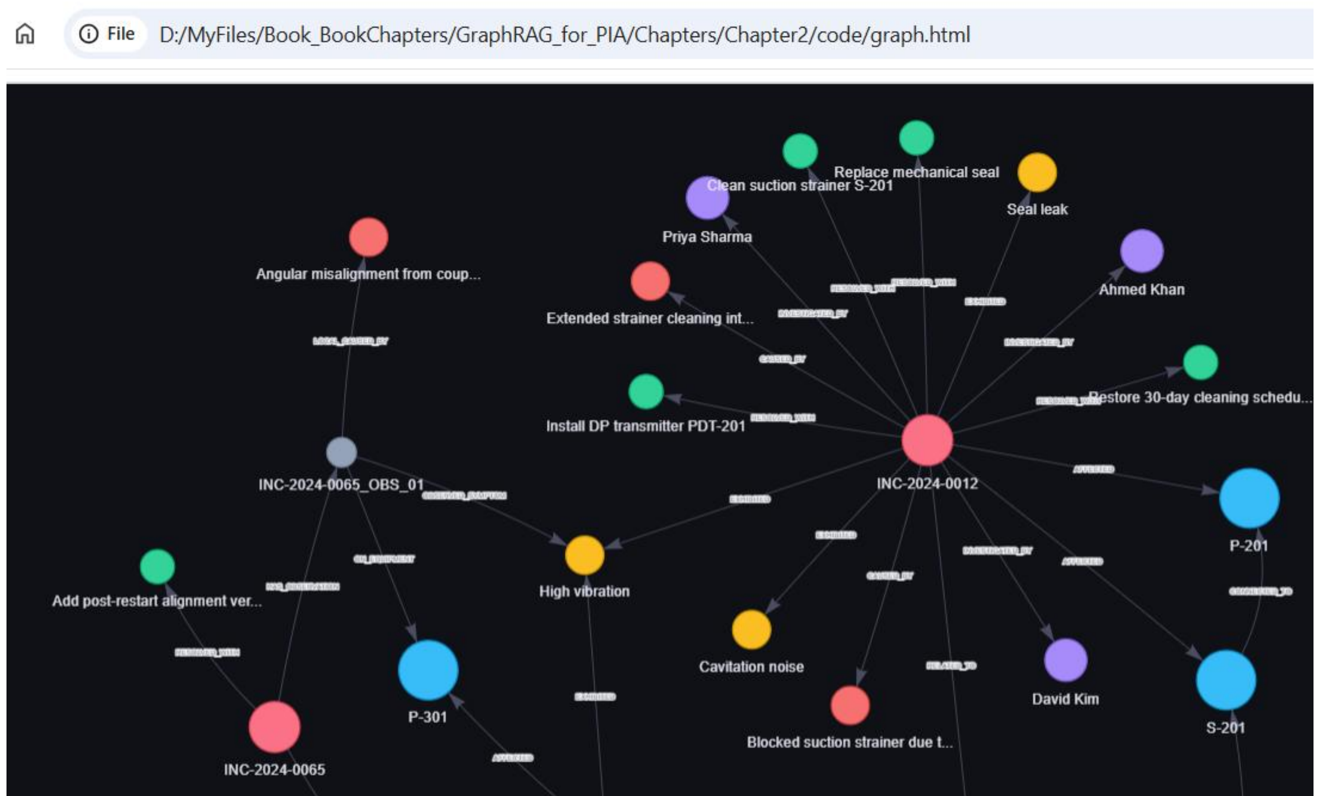
There is a simpler alternative to the Observation pattern: store the equipment tag as a property on the EXHIBITED relationship edge itself, like (Incident)-[:EXHIBITED {equipment_tag: "P-301"}]->(High vibration). The disambiguation query becomes: MATCH (i)-[r:EXHIBITED]->(s) RETURN r.equipment_tag, s.name. You can also add caused_by and resolved_by as edge properties. This is simpler to build and produces a visually cleaner graph.



The trade-off is that equipment_tag is a string reference, not a graph link; to get the Equipment node's details alongside the symptom, you need a second MATCH on the tag. Additionally, when an edge accumulates many properties (equipment, cause, action, description, measured values), it starts behaving like a node, which is a sign that it should be one. We use the Observation pattern in this book because it scales better for complex incidents and demonstrates important graph modeling concepts. But the edge-property approach is a perfectly valid choice for simpler use cases.

2.6 Visualization

Seeing the graph visually brings everything together. The notebook uses *pyvis*[8] to generate an interactive HTML visualization where you can zoom, pan, drag nodes, and hover for property details. Each node type gets a distinct color. The visualization is particularly valuable for spotting patterns: you can immediately see that P-201 has two incident edges (repeat failure), that INC-0012 and INC-0058 are linked by a dashed RELATED_TO edge (cascading failure), and that the Observation nodes sit neatly between INC-0065 and its two equipment items (multi-equipment disambiguation). Go ahead and run the Notebook cell and then open the generated *graph.html* file in your browser as shown below.



Neo4j AuraDB includes a built-in browser interface (accessible at your AuraDB dashboard) where you can run Cypher queries and see results as an interactive graph. This is excellent for ad-hoc exploration during development.

[8] https://pyvis.readthedocs.io/en/latest/

Summary

In this chapter, you built your first knowledge graph from scratch. You learned the Cypher query language through process industry examples, constructed a mini troubleshooting graph with 3 equipment, 2 incidents, symptoms, causes, actions, and technicians, and queried it for equipment history, root cause analysis, and cascading failure chains.

But building a graph by hand (node by node, relationship by relationship) is tedious. For 2 incidents it took us dozens of Cypher statements. For 10 incidents, it would take hundreds. For a real plant with thousands of incident reports, it would be impossible. That's exactly what Chapter 3 solves. We'll use an LLM to read each incident report and automatically extract all entities, observations, and relationships using OpenAI's structured output feature. The Pydantic ontology we define will enforce our schema at extraction time, and the ingestion script will MERGE everything into Neo4j, scaling from our 3-equipment mini graph to a rich 80+ node graph from 10 reports in minutes.

Chapter 3

LLM-Powered Entity Extraction

In chapter 2 we built a mini troubleshooting graph by hand: node by node, relationship by relationship. For two incidents and three pieces of equipment, this was a manageable exercise. But imagine doing it for ten reports. Or a hundred. Or the thousands of incident reports accumulated over a refinery's lifetime. Manual construction simply doesn't scale. The good news is that LLMs are remarkably good at reading unstructured text and extracting structured information. Given an incident report and a well-defined schema, an LLM can identify every piece of equipment mentioned, every symptom observed, every root cause diagnosed, and every corrective action taken, and return them as a structured, typed object that we can directly load into our graph database. Our job is to define the schema (what to extract) and the extraction rules (how to extract it), and the LLM does the heavy lifting.

In this chapter, you will design an ontology using Pydantic models, extract entities from all 10 incident reports using OpenAI's structured output, walk through a complete extraction result field by field to understand exactly what nodes and edges are created, ingest everything into Neo4j, and verify the result.

Specifically, the following topics are covered

- The ontology as Pydantic models: defining entity types, constrained values, and the Observation pattern
- A complete extraction walkthrough: reading the raw JSON field by field and mapping each piece to nodes and edges
- The extraction prompt: what the schema defines (structure) vs what the prompt defines (behavior)
- Batch extraction and ingestion: processing all 10 reports

3.1 Designing the Graph Schema

Before we can extract anything, we need to define precisely what entities and relationships our knowledge graph should contain. In Chapter 2, we introduced the entity types informally. Now we formalize them using Pydantic models (define in the file *schema.py*).

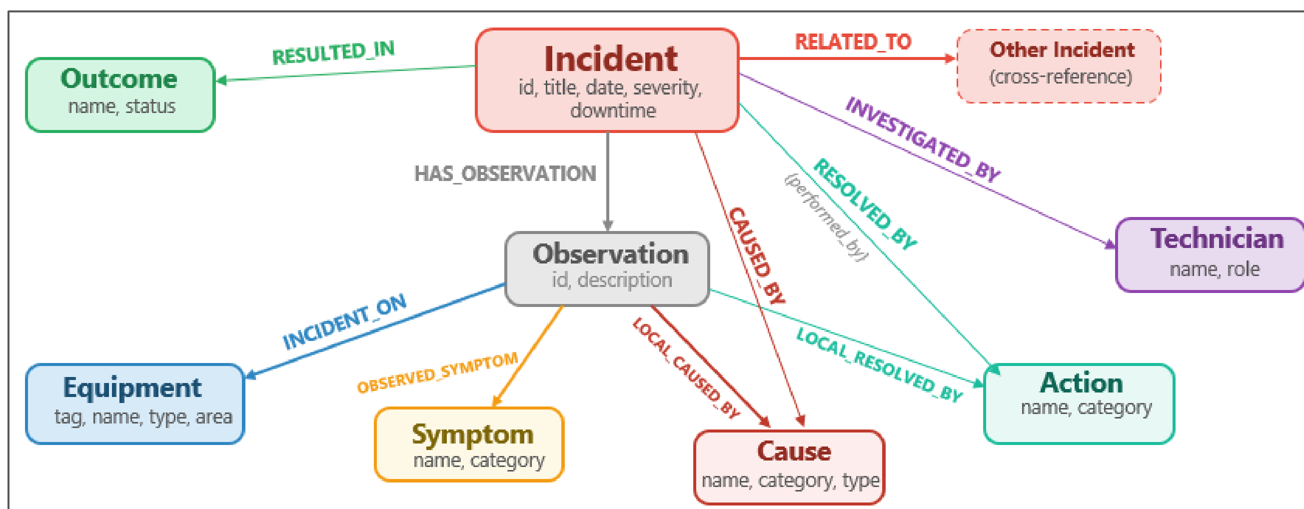


Figure 3.1: The Complete Ontology — Entity Types, Relationships

Below we show how the model for Equipment node is defined:

```
# imports
from enum import Enum
from typing import Optional
from pydantic import BaseModel, Field

# specify the different equipment types
class EquipmentType(str, Enum):
    CENTRIFUGAL_PUMP = "Centrifugal Pump"
    RECIPROCATING_COMPRESSOR = "Reciprocating Compressor"
    HEAT_EXCHANGER = "Heat Exchanger"
    # ... 9 more types

# define the properties of Equipment node
class Equipment(BaseModel):
    tag: str = Field(description="Tag number as primary ID, e.g. 'P-201'")
    name: str = Field(description="Full name, e.g. 'Feed Pump P-201'")
```

```
    equipment_type: EquipmentType   # enum with 12 allowed values as defined above
    area: str = Field(description="Plant unit (e.g., 'Crude Distillation Unit')")
```

The EquipmentType enum constrains the LLM to exactly 12 equipment categories. Without this, the LLM might undesirably produce “pump,” “centrifugal pump,” “rotating pump,” and “feed pump” as four different types. Another important model in our schema is the Observation model (the junction node we defined in the previous chapter) which is defined as below:

```
# An observation ties ONE equipment to ONE symptom within an incident. It can optionally
# carry its own local causes and actions when the report clearly attributes them to this specific
# equipment-symptom pair.
class Observation(BaseModel):
  observation_id: str = Field(description="Unique ID: <incident_id>_OBS_<seq>, e.g., 'INC-2024-0065_OBS_01'")
  equipment_tag: str = Field(description="Tag of the equipment this observation is about")
  symptom_name: str = Field(description="Canonical symptom name observed on this equipment")
  description: Optional[str] = Field(None, description="Brief description with measured values if available")
  local_causes: Optional[list[str]] = Field(None, description="Names of causes specific to THIS equipment-
                                      symptom pair (not incident-wide causes)")
  local_actions: Optional[list[str]] = Field(None, description="Names of actions specific to THIS equipment (not
                                      incident-wide actions)")
```

And the top-level IncidentExtraction model ties everything together:

```
# Complete extraction from one incident report using the Observation pattern
class IncidentExtraction(BaseModel):
    # The incident itself
    incident_id: str = Field(description="From report header (e.g., 'INC-2024-0012')")
    title: str = Field(description="Short descriptive title")
    date: str = Field(description="ISO format YYYY-MM-DD")
    severity: Severity
    downtime_hours: float = Field(description="Hours of downtime.")
    unit: str = Field(description="Plant unit name")
    summary: Optional[str] = Field(None, description="One-sentence summary of the incident")

    # Canonical entities (shared across incidents via MERGE)
    equipment: list[Equipment] = Field(description="ALL equipment mentioned in the report")
    symptoms: list[Symptom] = Field(description="ALL symptoms observed (canonical names)")
    causes: list[Cause] = Field(description="ALL causes — both primary and contributing")
    actions: list[Action] = Field(description="ALL corrective actions taken")
    outcomes: list[Outcome] = Field(description="Outcomes of the incident")
    technicians: list[Technician] = Field(description="ALL personnel involved")
```

```
# Observations: the equipment-symptom pairings
observations: list[Observation] = Field(description="One observation per equipment-symptom pair. CRITICAL:
                                        every symptom must be linked to the specific equipment that exhibited it.")

# Incident-level cause/action attribution
incident_level_causes: Optional[list[str]] = Field(None, description="Names of causes that apply to
                                        the whole incident (shared/global), not to a specific observation")
incident_level_actions: Optional[list[str]] = Field(None, description="Names of actions that apply to
                                        the whole incident (e.g., procedure changes)")

# Cross-references
related_incidents: Optional[list[str]] = Field(None, description="IDs of related incidents referenced in
                                        the report (e.g., 'INC-2024-0023')")
```

Alternative Design to Keep Extraction Consistent as the Graph Grows

In our current schema, all the entity fields are constrained with pre-specified values. Some fields have a genuinely closed set of values, such as equipment types (there are only so many kinds of rotating and static equipment in a refinery), severity levels (Low/Medium/High/Critical), and outcome statuses (Resolved/Partially Resolved/Monitoring/Pending). For these, Pydantic Enums are the right choice: they prevent the LLM from inventing variants and keep the graph clean.

But fields like cause category, symptom name, and action category are open-ended. You cannot anticipate every possible failure mode before you have seen the reports. If you constrain these with rigid Enums, the LLM will silently force-fit novel values into the closest available option (a "PLC logic error" might become "Operational" or "Design") neither of which is accurate.

A potential alternative is to use Enums for closed sets and plain strings with descriptive guidance for open-ended fields:

```
# Open-ended → str with guidance (flexible)
class Cause(BaseModel):
    name: str
    category: str = Field(description="e.g., Fouling, Corrosion,  "Mechanical Wear,
                          Misalignment, Human Error, "Operational, or other as appropriate")
```

The Field description guides the LLM toward consistent naming without constraining it.

When the LLM encounters a PLC logic error, it can create a "Software" category instead of being forced into an ill-fitting enum value. However, there remains a practical issue: consistency over time. After processing 50 reports, your graph might contain 25 distinct cause names. When report #51 describes the same failure mode that report #30 introduced as "Software Logic Failure," the LLM (not knowing that name already exists) might call it "Control System Error." Now you have two nodes for the same concept.

A practical solution is retrieval-augmented extraction: before extracting a new report, query your graph for all existing entity names and inject them into the system prompt:

```
# Fetch current graph vocabulary
existing_causes = query("MATCH (c:Cause) RETURN DISTINCT c.name")
existing_symptoms = query("MATCH (s:Symptom) RETURN DISTINCT s.name")

# Inject into the extraction prompt
prompt = f"""
EXISTING ENTITIES (reuse exact names when applicable):
Causes: {existing_causes}
Symptoms: {existing_symptoms}

If a cause/symptom in this report matches an existing entity,
use the EXACT same name. If genuinely new, create a new name.
"""
```

This creates a virtuous feedback loop: the graph's current vocabulary informs the extraction of new documents, which in turn extends the vocabulary for future extractions. Combined with Neo4j's MERGE (which deduplicates on name), this keeps the graph clean and consistent as it grows.

3.2 The Extraction LLM

With our Pydantic ontology defined, we use OpenAI’s structured output feature to extract entities from an incident report as shown below (defined in script *extract_entities.py*):

```
# function to extract entities from a single report using an LLM  and return a typed Pydantic object
def extract_from_report(report_text: str) -> IncidentExtraction:
  """Extract entities using structured output """
  response = client.responses.parse(
    model="gpt-5.2",
    instructions=EXTRACTION_SYSTEM_PROMPT,
    input=f"Extract all entities and observations:\n\n{report_text}",
    text_format=IncidentExtraction,)

  return response.output_parsed
```

The *text_format* enforces our Pydantic schema as shown in Figure 3,2 below. The API call will not produce output that violates the schema: enum values are constrained and required fields are always present.

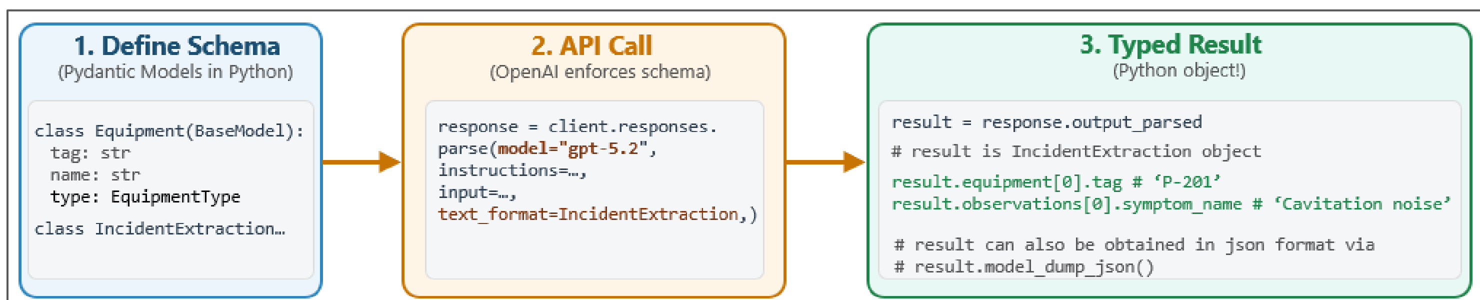


Figure 3.2: LLM Structured Output -> From Pydantic Model to Typed Object[9]

Extraction Prompt

While the Pydantic schema defines the structure (what fields exist), the system prompt defines the behavior (how to fill them). The prompt, EXTRACTION_SYSTEM_PROMPT, that we used in the API call is defines as below:

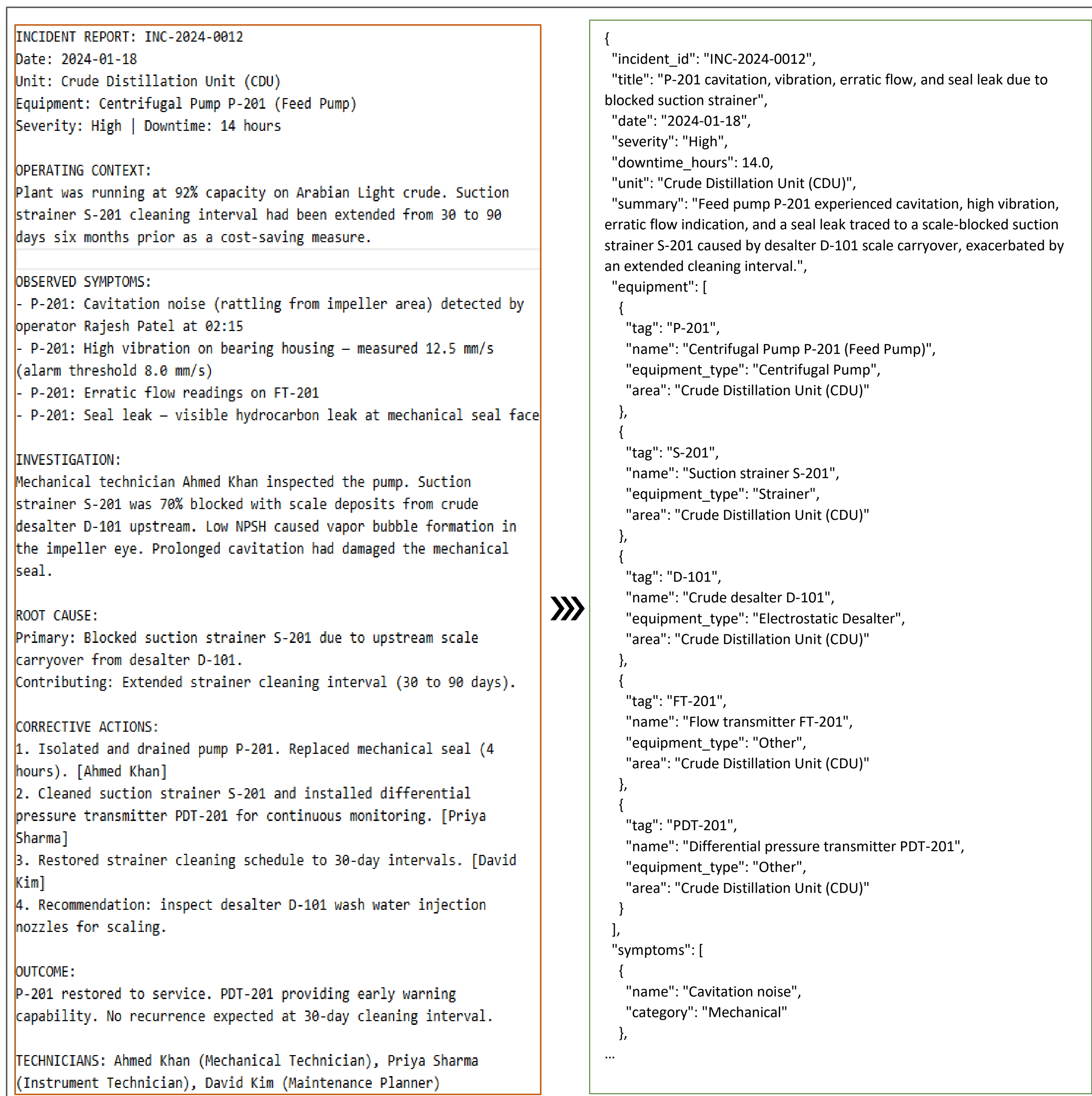
```
# the main instruction given to the LLM for entity extraction
EXTRACTION_SYSTEM_PROMPT = """You are a process industry knowledge graph extraction
engine. Given an incident report, extract ALL entities, observations, and relationships.

CORE CONCEPT — THE OBSERVATION:
An Observation ties ONE equipment to ONE symptom. In a multi-equipment incident,
```

[9] Notebook *ch3_llm_extraction.ipynb* shows example executions

```
each equipment-symptom pair gets its own Observation.

Example — coupled pump-motor incident:
  Observation 1: P-301 + High vibration (caused locally by angular misalignment)
  Observation 2: M-301 + High winding temperature (caused locally by excessive load)
  Incident-level action: Added post-restart alignment verification procedure

EXTRACTION RULES:

1. EQUIPMENT: Extract EVERY piece of equipment mentioned — primary equipment,
   secondary components (strainers, knockout drums, motors), and upstream/downstream
   equipment. Always use tag numbers (P-201, M-301, S-201, V-401).

2. OBSERVATIONS: Create one Observation per equipment-symptom pair.
   - observation_id format: <incident_id>_OBS_01, _OBS_02, etc.
   - Link local causes and actions when the report clearly attributes them
     to a specific equipment-symptom pair.
   - If a cause or action applies to the whole incident (shared), put it in
     incident_level_causes / incident_level_actions instead.

3. SYMPTOMS: Use canonicalized short phrases. Map report language:
   "abnormal noise" → "Cavitation noise"
   "fluctuating pressure" → "Erratic flow"
   "vibration alarm" → "High vibration"
   "motor running hot" → "High winding temperature"

4. CAUSES: Specific, present tense, include the component.
   "Blocked suction strainer due to upstream scale carryover" ✓
   "The strainer was blocked" ✗
   Mark each as "primary" or "contributing" in cause_type.

5. ACTIONS: Include performed_by (technician name) when the report specifies it.
   This will be stored as an edge property on RESOLVED_WITH in the graph.

6. DUAL-LEVEL ATTRIBUTION:
   - Observation-level: causes/actions specific to one equipment-symptom pair
   - Incident-level: shared causes/actions (e.g., procedure changes, QA failures)
   When in doubt, use incident-level.

7. CROSS-REFERENCES: If the report mentions another incident ID, add it to related_incidents.

8. OUTCOMES: Capture the final state after corrective actions."""
```

Extraction Result

Figure 3.3 below shows an example of how the output from the LLM Api call look like. In the next section, we will see how this output is ingested into our knowledge graph.



INCIDENT REPORT: INC-2024-0012
Date: 2024-01-18
Unit: Crude Distillation Unit (CDU)
Equipment: Centrifugal Pump P-201 (Feed Pump)
Severity: High | Downtime: 14 hours

OPERATING CONTEXT:
Plant was running at 92% capacity on Arabian Light crude. Suction strainer S-201 cleaning interval had been extended from 30 to 90 days six months prior as a cost-saving measure.

OBSERVED SYMPTOMS:
- P-201: Cavitation noise (rattling from impeller area) detected by operator Rajesh Patel at 02:15
- P-201: High vibration on bearing housing – measured 12.5 mm/s (alarm threshold 8.0 mm/s)
- P-201: Erratic flow readings on FT-201
- P-201: Seal leak – visible hydrocarbon leak at mechanical seal face

INVESTIGATION:
Mechanical technician Ahmed Khan inspected the pump. Suction strainer S-201 was 70% blocked with scale deposits from crude desalter D-101 upstream. Low NPSH caused vapor bubble formation in the impeller eye. Prolonged cavitation had damaged the mechanical seal.

ROOT CAUSE:
Primary: Blocked suction strainer S-201 due to upstream scale carryover from desalter D-101.
Contributing: Extended strainer cleaning interval (30 to 90 days).

CORRECTIVE ACTIONS:
1. Isolated and drained pump P-201. Replaced mechanical seal (4 hours). [Ahmed Khan]
2. Cleaned suction strainer S-201 and installed differential pressure transmitter PDT-201 for continuous monitoring. [Priya Sharma]
3. Restored strainer cleaning schedule to 30-day intervals. [David Kim]
4. Recommendation: inspect desalter D-101 wash water injection nozzles for scaling.

OUTCOME:
P-201 restored to service. PDT-201 providing early warning capability. No recurrence expected at 30-day cleaning interval.

TECHNICIANS: Ahmed Khan (Mechanical Technician), Priya Sharma (Instrument Technician), David Kim (Maintenance Planner)


```
{
  "incident_id": "INC-2024-0012",
  "title": "P-201 cavitation, vibration, erratic flow, and seal leak due to blocked suction strainer",
  "date": "2024-01-18",
  "severity": "High",
  "downtime_hours": 14.0,
  "unit": "Crude Distillation Unit (CDU)",
  "summary": "Feed pump P-201 experienced cavitation, high vibration, erratic flow indication, and a seal leak traced to a scale-blocked suction strainer S-201 caused by desalter D-101 scale carryover, exacerbated by an extended cleaning interval.",
  "equipment": [
   {
    "tag": "P-201",
    "name": "Centrifugal Pump P-201 (Feed Pump)",
    "equipment_type": "Centrifugal Pump",
    "area": "Crude Distillation Unit (CDU)"
   },
   {
    "tag": "S-201",
    "name": "Suction strainer S-201",
    "equipment_type": "Strainer",
    "area": "Crude Distillation Unit (CDU)"
   },
   {
    "tag": "D-101",
    "name": "Crude desalter D-101",
    "equipment_type": "Electrostatic Desalter",
    "area": "Crude Distillation Unit (CDU)"
   },
   {
    "tag": "FT-201",
    "name": "Flow transmitter FT-201",
    "equipment_type": "Other",
    "area": "Crude Distillation Unit (CDU)"
   },
   {
    "tag": "PDT-201",
    "name": "Differential pressure transmitter PDT-201",
    "equipment_type": "Other",
    "area": "Crude Distillation Unit (CDU)"
   }
  ],
  "symptoms": [
   {
    "name": "Cavitation noise",
    "category": "Mechanical"
   },
...
```


Figure 3.3: Extracting entities from an incident report

3.3 Ingestion into Graph

Alright, we are now in the final stage of our ingestion-to-extraction pipeline (Figure 3.4). The extracted entities (of the type shown in Figure 3.3) from each report need to be ingested into out Neo4j graph. Script *extract_entities.py* saves the Pydantic object for each incident report as a json file which is then ingested by the ingester script.

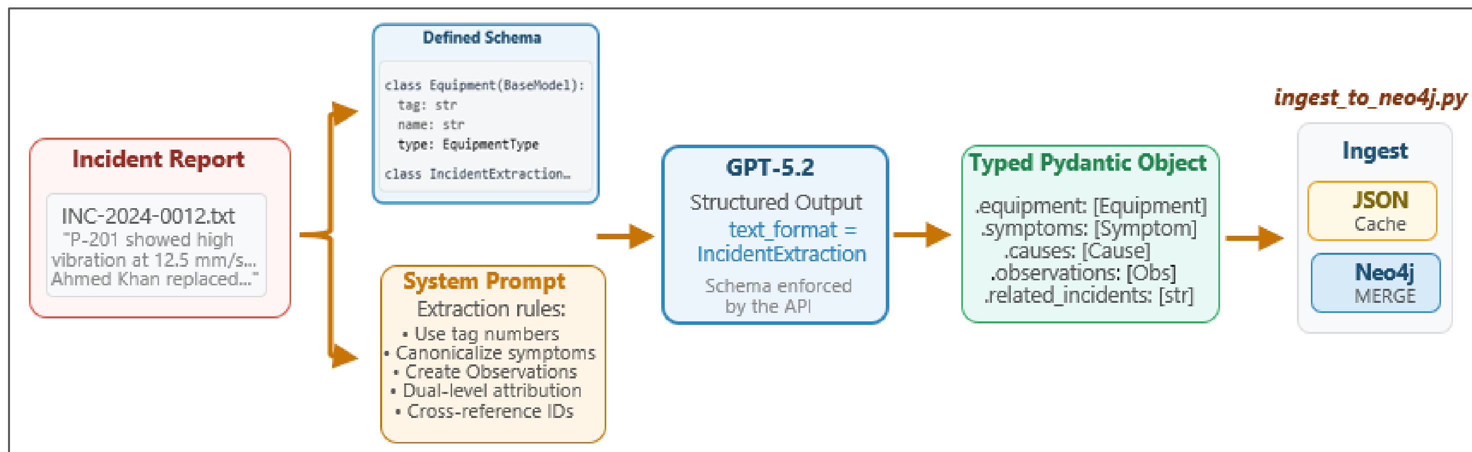


Figure 3.4: The extraction-to-ingestion pipeline

Consider the report INC-2024-0012 again. Figure 3.5 shows the resulting nodes and relationships created (/merged) by the ingester script based on the extracted entities shown in Figure 3.3. Similar nodes and relationships are created for the other reports as well; shared entities are deduplicated via MERGE (they become single nodes with multiple incoming edges).

If you see the ingester script, *ingest_to_neo4j.py*, you will find that it processes each JSON file through 6 groups of MERGE statements in the following order:

- **Incident node**: MERGE on incident_id, SET all properties.
- **Canonical entity nodes**: MERGE Equipment, Symptom, Cause, Action, Outcome, Technician. Also create INCIDENT_ON convenience edges.
- **Observation nodes and links**: For each Observation, create the node and link it to the Incident (HAS_OBSERVATION), Equipment (ON_EQUIPMENT), Symptom (OBSERVED_SYMPTOM), and optionally to Causes (LOCAL_CAUSED_BY) and Actions (LOCAL_RESOLVED_BY).
- **Incident-level links**: CAUSED_BY (shared causes), RESOLVED_BY (shared actions), RESULTED_IN (outcomes).

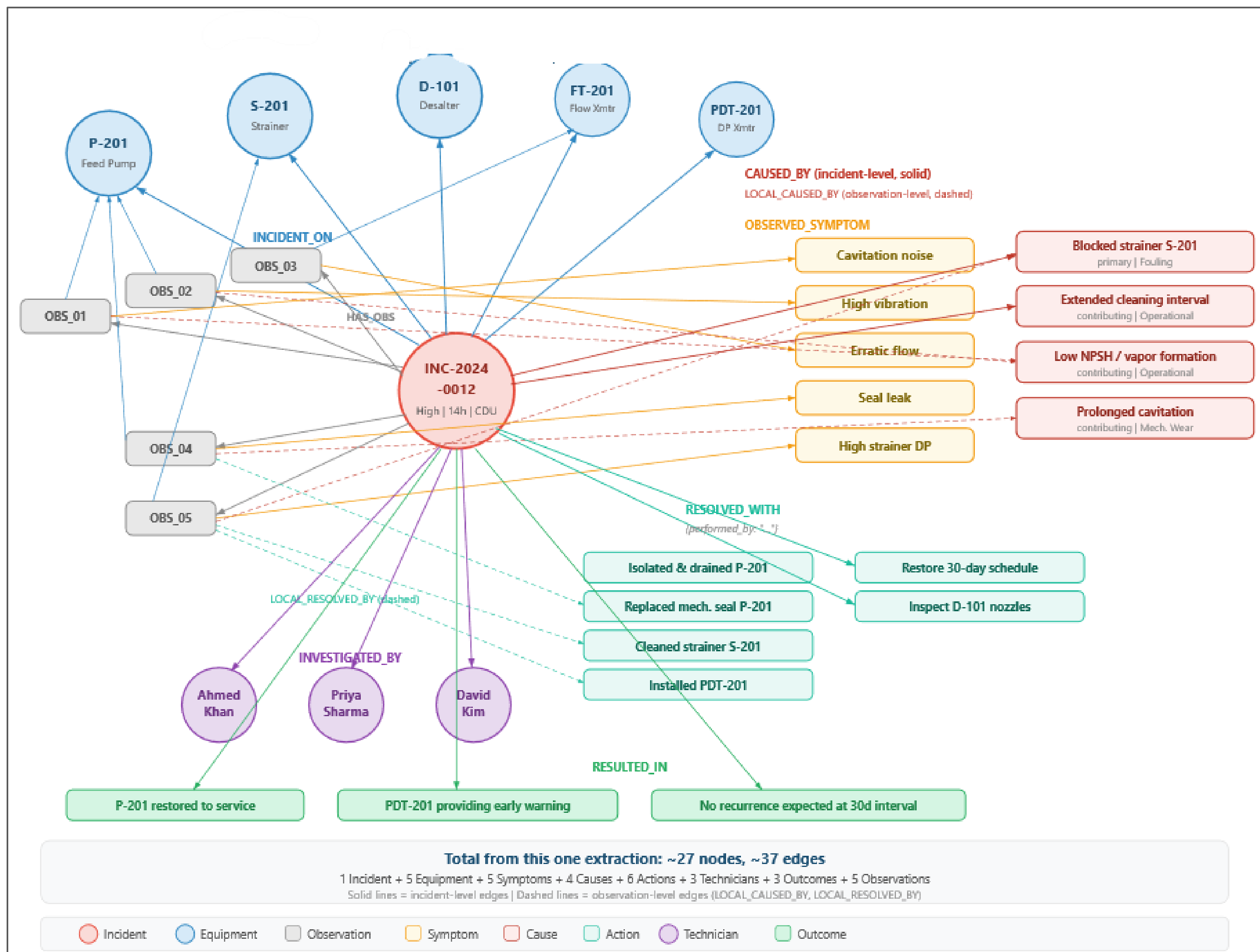


Figure 3.5: From extraction to Graph for report INC-2024-0012

- **Technician links**: INVESTIGATED_BY from Incident, PERFORMED_BY from Action.
- **Cross-references**: RELATED_TO between incidents.

Every statement uses MERGE, making the entire process idempotent. You can reprocess all 10 reports without creating duplicates. Uniqueness constraints on primary keys ensure that nodes are matched correctly. After you have finished populating your knowledge graph, you can visualize the graph using either of the mechanisms presented in the previous chapter.

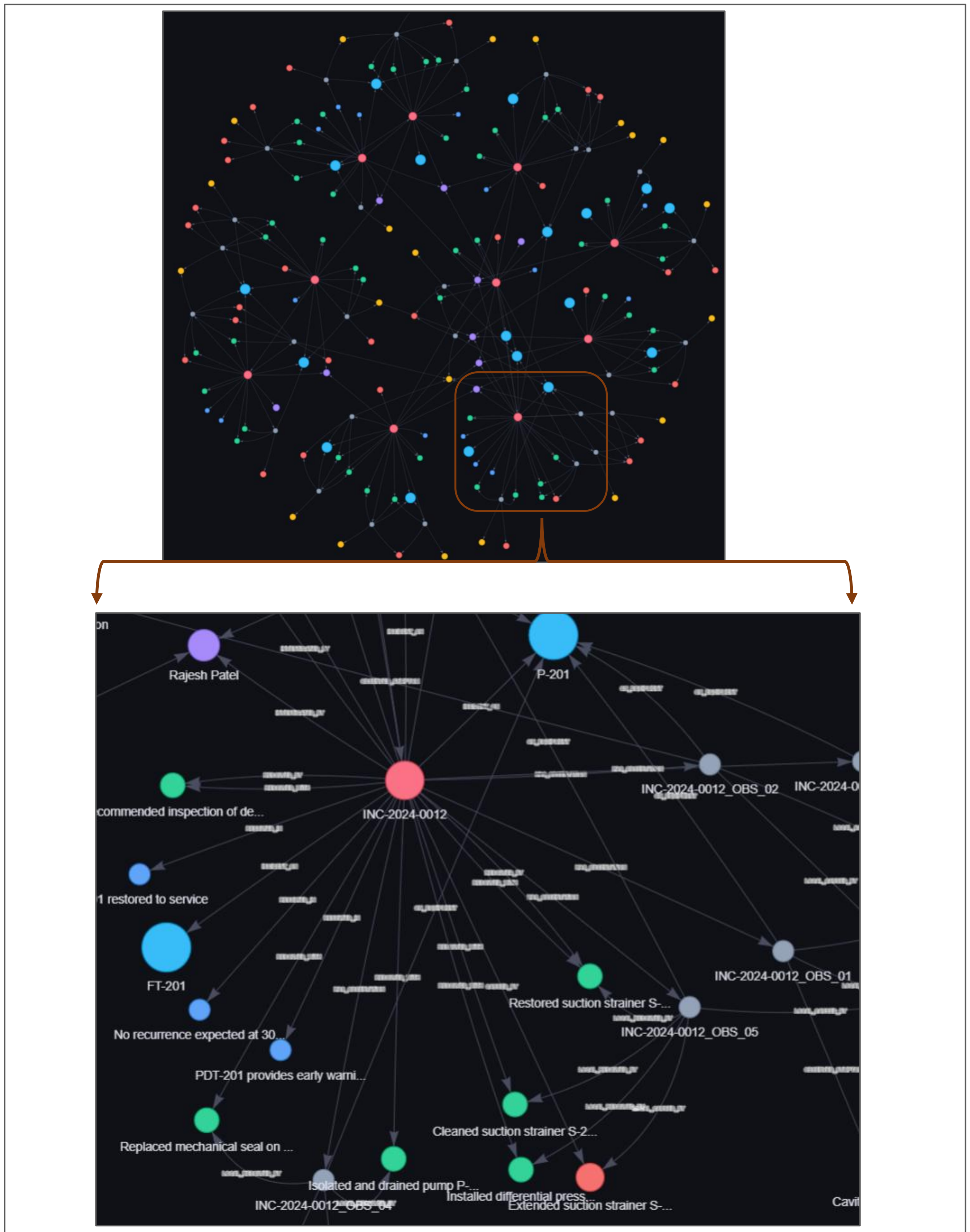


Figure 3.6: The complete graph with zoomed in view of INC-2024-0012 incident node

Summary

In this chapter, we automated what took dozens of manual Cypher statements in Chapter 2. By defining our ontology as Pydantic models and using OpenAI's structured output, we extracted entities from 10 incident reports very quickly and conveniently! The Observation pattern correctly handles multi-equipment incidents, and the MERGE-based ingestion ensures a clean, deduplicated graph. The graph now contains the complete troubleshooting knowledge from all 10 reports - equipment, symptoms, causes, actions, technicians, outcomes, and cross-references - all connected by relationships that enable the multi-hop, cross-document queries we discussed in Chapter 1.

In Chapter 4, we will put this graph to work. We'll build the GraphRAG query pipeline that lets a process engineer ask natural language questions and receive grounded, multi-hop answers. We'll explore three different querying approaches and build a hybrid retrieval system that combines graph traversal with vector search.

Chapter 4

GraphRAG Query Pipeline

By now, you have done all the hard work to get your knowledge graph populated and ready. Now comes the payoff: letting a process engineer ask questions in plain English and getting back grounded, multi-hop answers that cite specific incident IDs, equipment tag numbers, and technician names. This is GraphRAG: the combination of a knowledge graph with LLM-powered querying.

This chapter covers three different querying approaches (each with different trade-offs), introduces hybrid retrieval that combines graph queries with vector search, provides a systematic head-to-head comparison of GraphRAG vs simple RAG, and walks through the architectural decisions you'll need to make when building your own system. Specifically, the following topics are covered:

- **The three-step pipeline**: how a question becomes a Cypher query and then becomes an answer
- **Approach A**: LangChain GraphCypherQAChain: fast prototyping in ~10 lines
- **Approach B**: Raw OpenAI SDK: full control, no framework dependency
- **Approach C:** Pre-defined templates: deterministic, production-safe

4.1 How GraphRAG Querying Works

Every GraphRAG query goes through the following three steps:

1. **Question → Cypher**: An LLM reads the user's natural language question along with the Neo4j database schema (node types, relationship types, property names) and generates a Cypher query to retrieve the relevant data.

2. **Execute on Neo4j**: The generated Cypher query runs against the graph database. Unlike vector search (which returns text chunks), graph execution returns structured records, i.e, rows with specific fields like incident_id, equipment_tag, cause_name.

3. **Answer Synthesis**: A second LLM call reads the structured records along with the original question and produces a grounded, natural language answer. Because the data is structured (not free text), the LLM is far less likely to hallucinate.

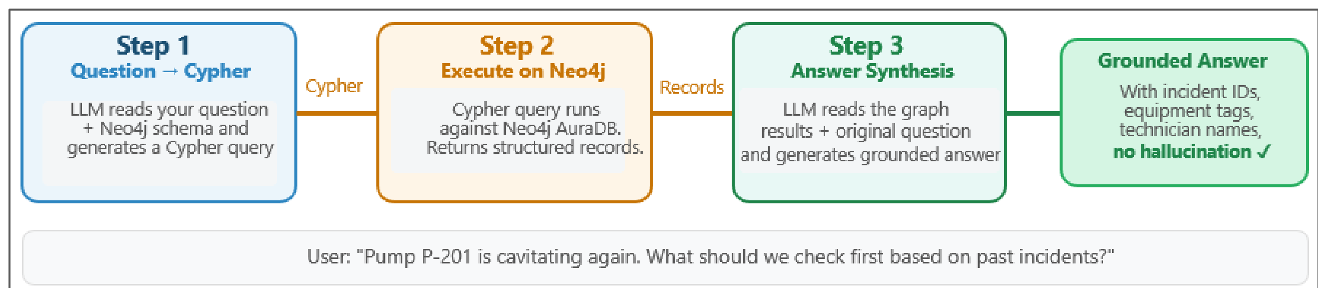


Figure 4.1: The Three-Step GraphRAG Query Pipeline

The key difference from simple RAG is in step 1: instead of embedding the question and doing vector similarity search, we generate a Cypher query that can traverse any number of relationship hops, aggregate across all incidents, and return precisely the data needed.

Let us now look at a few different approaches that you may use to implement your query pipeline. Of the shown approaches in Figure 4.2, we will study Approach A in detail in this chapter; approaches B and C are discussed conceptually[10].

[10] The Notebook *ch4_query_pipeline.ipynb* shows code implementations for all three approaches.

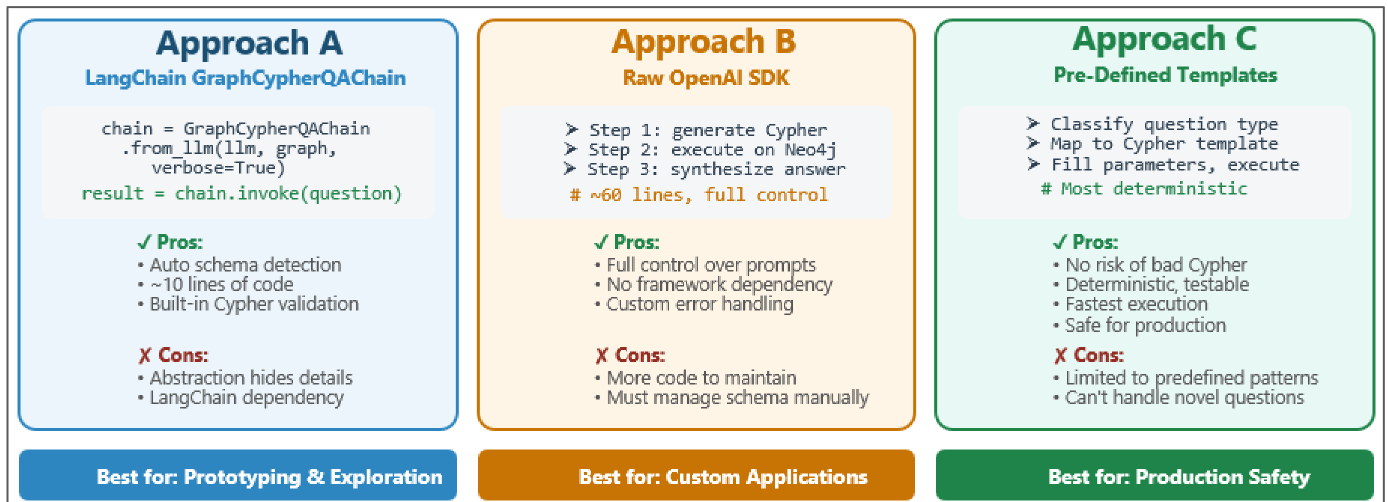


Figure 4.2: Three GraphRAG Querying Approaches

4.2 Approach A: LangChain GraphCypherQAChain

LangChain's GraphCypherQAChain is a fast and convenient path to a working GraphRAG pipeline. It handles graph schema detection, Cypher generation, execution, and answer synthesis via built-in modules; as you will see later, the entire three-step pipeline into a single callable object. Let's walk through the different implementation steps.

Step: Connect LangChain to Neo4j Graph

First, we create a Neo4jGraph object. This automatically reads the database schema, i.e., node labels, their properties, relationship types, and how they connect. This schema is later injected into the Cypher generation prompt, so the LLM always knows what's in your graph.

```
# relevant imports got Approach A[11]
from langchain_neo4j import Neo4jGraph, GraphCypherQAChain
from langchain_openai import ChatOpenAI
from langchain_core.prompts import PromptTemplate

# create Neo4jGraph object; LangChain auto-reads the Neo4j schema!
lc_graph = Neo4jGraph(
    url=os.getenv('NEO4J_URI'),
    username=os.getenv('NEO4J_USERNAME'), password=os.getenv('NEO4J_PASSWORD'),)
```

[11] pip install langchain langchain-neo4j langchain-openai

```
# See what LangChain detected:
print(lc_graph.schema[:500])

>>> Node properties:
Equipment {area: STRING, name: STRING, tag: STRING, equipment_type: STRING}
Incident {date: STRING, downtime_hours: FLOAT, incident_id: STRING, severity: STRING, title:
STRING, unit: STRING, summary: STRING}
Symptom {name: STRING, category: STRING}
Cause {name: STRING, category: STRING, cause_type: STRING}
Action {name: STRING, category: STRING}
Technician {name: STRING, role: STRING}
Observation {description: STRING, observation_id: STRING}
Outcome {name: STRING, status: STRING}
Relationship properties:
RESOLVED_WITH {performed_by: STRING}
The relationships:
(:Incident)-[:CAUSED_BY]->(:Cause)
(:Incident)-[:RESOLVED_WITH]->(:Action)
(:Incident)-[:INVESTIGATED_BY]->(:Technician)
(:Incident)-[:HAS_OBSERVATION]->(:Observation)
(:Incident)-[:INCIDENT_ON]->(:Equipment)
(:Incident)-[:RESOLVED_BY]->(:Action)
(:Incident)-[:RESULTED_IN]->(:Outcome)
(:Incident)-[:RELATED_TO]->(:Incident)
(:Observation)-[:ON_EQUIPMENT]->(:Equipment)
(:Observation)-[:OBSERVED_SYMPTOM]->(:Symptom)
(:Observation)-[:LOCAL_CAUSED_BY]->(:Cause)
(:Observation)-[:LOCAL_RESOLVED_BY]->(:Action)
```

This is one of LangChain's biggest advantages: you don't need to write the schema manually. It introspects your database and injects the schema into the Cypher generation prompt automatically. If you add new node types or properties later, call *lc_graph.refresh_schema()* and the prompt updates.

Step: Write the Cypher Generation Prompt

While LangChain provides default prompts, custom prompts dramatically improve Cypher quality for domain-specific applications. Our Cypher generation prompt includes a few query pattern templates that serve as few-shot examples: equipment history, symptom diagnosis, expert finding, cascading failures, and downtime aggregation; these 'teach' the LLM how to traverse our ontology.

```
# The prompt includes query pattern templates that guide the LLM toward correct Cypher
CYPHER_PROMPT = PromptTemplate(
    input_variables=['schema', 'question'],
    template="""You are a Cypher expert for a process plant troubleshooting knowledge graph.
Generate ONLY a Cypher query; no explanation, no markdown.

Schema:
{schema}

KEY PATTERNS (uses Observation nodes for equipment-symptom pairing):

1. Equipment history:
  MATCH (e:Equipment {{tag: "X"}})<-[:ON_EQUIPMENT]-(ob:Observation)
      <-[:HAS_OBSERVATION]-(i:Incident)
  MATCH (ob)-[:OBSERVED_SYMPTOM]->(s:Symptom)
  OPTIONAL MATCH (ob)-[:LOCAL_CAUSED_BY]->(c:Cause)
  RETURN i.incident_id, i.date, s.name, c.name

2. Symptom diagnosis (aggregated across all incidents):
  MATCH (s:Symptom)<-[:OBSERVED_SYMPTOM]-(ob:Observation)
      -[:ON_EQUIPMENT]->(e:Equipment)
  WHERE toLower(s.name) CONTAINS "keyword"
  MATCH (ob)<-[:HAS_OBSERVATION]-(i:Incident)
  OPTIONAL MATCH (ob)-[:LOCAL_CAUSED_BY]->(c:Cause)
  RETURN c.name, count(DISTINCT i) AS occurrences, collect(DISTINCT e.tag)

3. Multi-equipment disambiguation:
  MATCH (i:Incident {{incident_id: "X"}})-[:HAS_OBSERVATION]->(ob:Observation)
      -[:ON_EQUIPMENT]->(e:Equipment)
  MATCH (ob)-[:OBSERVED_SYMPTOM]->(s:Symptom)
  OPTIONAL MATCH (ob)-[:LOCAL_CAUSED_BY]->(c:Cause)
  RETURN e.tag, s.name, c.name

4. Expert finder:
  MATCH (t:Technician)<-[:INVESTIGATED_BY]-(i:Incident)-[:INCIDENT_ON]->(e:Equipment)
  WHERE e.equipment_type CONTAINS "keyword"
  RETURN t.name, t.role, count(DISTINCT i) AS experience

5. Cascading failures:
  MATCH (i1:Incident)-[:RELATED_TO*1..3]->(i2:Incident)-[:INCIDENT_ON]->(e:Equipment)
  RETURN i1.incident_id, i2.incident_id, e.tag

6. Downtime aggregation:
```

```
  MATCH (c:Cause)<-[:LOCAL_CAUSED_BY]-(ob)<-[:HAS_OBSERVATION]-(i:Incident)
  RETURN c.category, sum(DISTINCT i.downtime_hours), count(DISTINCT i)

7. Action history with performers (edge property on RESOLVED_WITH):
  MATCH (i:Incident)-[r:RESOLVED_WITH]->(a:Action)
  WHERE a.name CONTAINS "keyword"
  RETURN i.incident_id, a.name, r.performed_by

RULES:
- Use OPTIONAL MATCH for paths that might not exist
- Always traverse through Observation nodes for equipment-symptom queries
- For equipment, match on tag property (e.g., "P-201")
- LIMIT to 25 unless aggregating

Question: {question}""")
```

Step: Write the Answer Synthesis Prompt

The QA prompt tells the LLM how to format answers for process engineers; it enforces process industry conventions: cite incident IDs, use equipment tag numbers, recommend technicians by name, note repeat failure patterns, and never hallucinate. The key instruction is “Answer using ONLY the knowledge graph data below.” This grounds the LLM’s answer in the structured records from Neo4j, preventing hallucination.

```
# Custom Answer Synthesis Prompt
QA_PROMPT = PromptTemplate(
    input_variables=['context', 'question'],
    template="""You are a senior process plant troubleshooting assistant.
Answer using ONLY the knowledge graph data below.

RULES:
- Cite incident IDs (e.g., INC-2024-0012) when referencing events
- Use equipment tag numbers (P-201, M-301, not descriptive names)
- When multi-equipment incidents are involved, clearly state which
  equipment showed which symptom and what the local cause was
- Recommend technicians by name when the data supports it
- Note patterns: repeat failures, cascading issues
- If data is insufficient, say so; never hallucinate

Graph Data:
{context}
```

```
Question: {question}

Provide a detailed, actionable answer:"""
)
```

Step: Build and Test the Chain

Now we wire everything together.

```
# GraphCypherQAChain wires everything together[12]:
# Question → Cypher generation → Neo4j execution → Answer synthesis
chain_a = GraphCypherQAChain.from_llm(
   llm=llm,
   graph=lc_graph,
   verbose=True,           # IMPORTANT: shows generated Cypher!
   allow_dangerous_requests=True,
   return_intermediate_steps=True, # returns Cypher + raw results
   cypher_prompt=CYPHER_PROMPT,
   qa_prompt=QA_PROMPT,
   top_k=25,)
```

That's it, the entire pipeline in one call. Let's test it:

```
# Test: Equipment history[13]
result = chain_a.invoke({'query': 'What happened to pump P-201?'})
print(result['result'])

>>> Pump **P-201** has had **repeat high-vibration events linked to suction restriction at strainer S-201**, driven by **upstream solids/scale carryover from desalter D-101**, with one event escalating into **cavitation and a mechanical seal leak**.

## Incident history and what happened (by equipment)

### 1) P-201 cavitation + vibration + seal leak due to blocked S-201 (INC-2024-0012)
- **P-201 symptoms:** cavitation noise, **high vibration**, **erratic flow**, and **mechanical seal leak** (INC-2024-0012).
  - Observed vibration: **12.5 mm/s** vs alarm threshold **8.0 mm/s**.
  ...
```

[12] *top_k* defines the maximum number of results to return from the graph query

[13] See complete answer in the chapter's Jupyter Notebook

With verbose=True, you'll see the generated Cypher printed before the answer. For this question, the LLM should generate something like:

```
# [Generated Cypher]:
MATCH (e:Equipment {tag:"P-201"})<-[:ON_EQUIPMENT]-(ob:Observation)<-
[:HAS_OBSERVATION]-(i:Incident)
OPTIONAL MATCH (ob)-[:OBSERVED_SYMPTOM]->(s:Symptom)
OPTIONAL MATCH (ob)-[:LOCAL_CAUSED_BY]->(lc:Cause)
OPTIONAL MATCH (ob)-[:LOCAL_RESOLVED_BY]->(la:Action)
OPTIONAL MATCH (i)-[:CAUSED_BY]->(ic:Cause)
OPTIONAL MATCH (i)-[rw:RESOLVED_WITH]->(ra:Action)
OPTIONAL MATCH (i)-[:RESULTED_IN]->(o:Outcome)
OPTIONAL MATCH (i)-[:INVESTIGATED_BY]->(t:Technician)
RETURN
  i.incident_id AS incident_id,
  i.date AS date,
  i.title AS title,
  i.severity AS severity,
  i.downtime_hours AS downtime_hours,
  collect(DISTINCT s.name) AS symptoms,
  collect(DISTINCT lc.name) AS local_causes,
  collect(DISTINCT ic.name) AS incident_causes,
  collect(DISTINCT la.name) AS local_actions,
  collect(DISTINCT {action: ra.name, performed_by: rw.performed_by}) AS
resolved_with_actions,
  collect(DISTINCT o.name) AS outcomes,
  collect(DISTINCT t.name) AS technicians,
  collect(DISTINCT ob.description) AS observations
ORDER BY date DESC
LIMIT 25;
```

4.3 Approach B: Custom Chain using OpenAI SDK

You don't have to use LangChain's GraphCypherQAChain to implement a GraphRAG query pipeline. You could build one using only OpenAI SDK; however, now you will have to manage injection of the knowledge graph schema into the LLM manually as shown below.

Step: Generate Cypher

```
# schema is provided as a string in the instructions
cypher_response = client.responses.create(
      model='gpt-5.2',
      instructions=f"""You are a Cypher query expert. Generate ONLY a Cypher query for this Neo4j graph.
                      No explanation. The graph uses Observation nodes:
                       Incident→HAS_OBSERVATION→Observation→ON_EQUIPMENT→Equipment.
                       Observation→OBSERVED_SYMPTOM→Symptom.
                       Observation→LOCAL_CAUSED_BY→Cause.
                       Schema: {schema_str}""",
      input=question,)
  cypher = cypher_response.output_text
```

Step: Execute on Neo4j

```
# fetch records from knowledge graph
with driver.session() as session:
   records = [dict(r) for r in session.run(cypher)]
```

Step: Synthesize Answer

```
# generate final answer for the user
answer_response = client.responses.create(
      model='gpt-5.2',
      instructions='You are a process plant troubleshooting assistant. Answer using ONLY the provided
                    data. Cite incident IDs and equipment tags. Never hallucinate.',
      input=f'Question: {question}\n\nGraph data:\n{json.dumps(records, indent=2, default=str)}
            \n\nProvide a detailed answer:')
 answer = answer_response.output_text
```

4.4 Approach C: Pre-Defined Templates

Approaches A and B both let the LLM generate Cypher freely. In production, this creates risk: a malformed query could fail, or worse, a cleverly crafted question could inject destructive Cypher. Approach C eliminates this risk entirely. The idea is simple: instead of generating Cypher, the LLM only classifies the question into a category and extracts required parameters. The actual Cypher is a pre-approved template. Let's briefly understand how this approach may be implemented.

Step: Define Templates

```
# Each template has a name, description (for classification), and parameterized Cypher
QUERY_TEMPLATES = {
    'equipment_history': {
        'description': 'What happened to a specific equipment?',
        'params': ['equipment_tag'],
        'cypher': '''
            MATCH (e:Equipment {tag: $equipment_tag})
                <-[:ON_EQUIPMENT]-(ob)<-[:HAS_OBSERVATION]-(i)
            MATCH (ob)-[:OBSERVED_SYMPTOM]->(s:Symptom)
            RETURN i.incident_id, s.name, i.date
        '''
    },
    'action_history': {
        'description': 'Show all incidents with a specific action',
        'params': ['action_keyword'],
        'cypher': '''
            MATCH (i:Incident)-[r:RESOLVED_WITH]->(a:Action)
            WHERE toLower(a.name) CONTAINS toLower($action_keyword)
            RETURN i.incident_id, a.name, r.performed_by
        '''
    },
    # ...more templates in the notebook
}
```

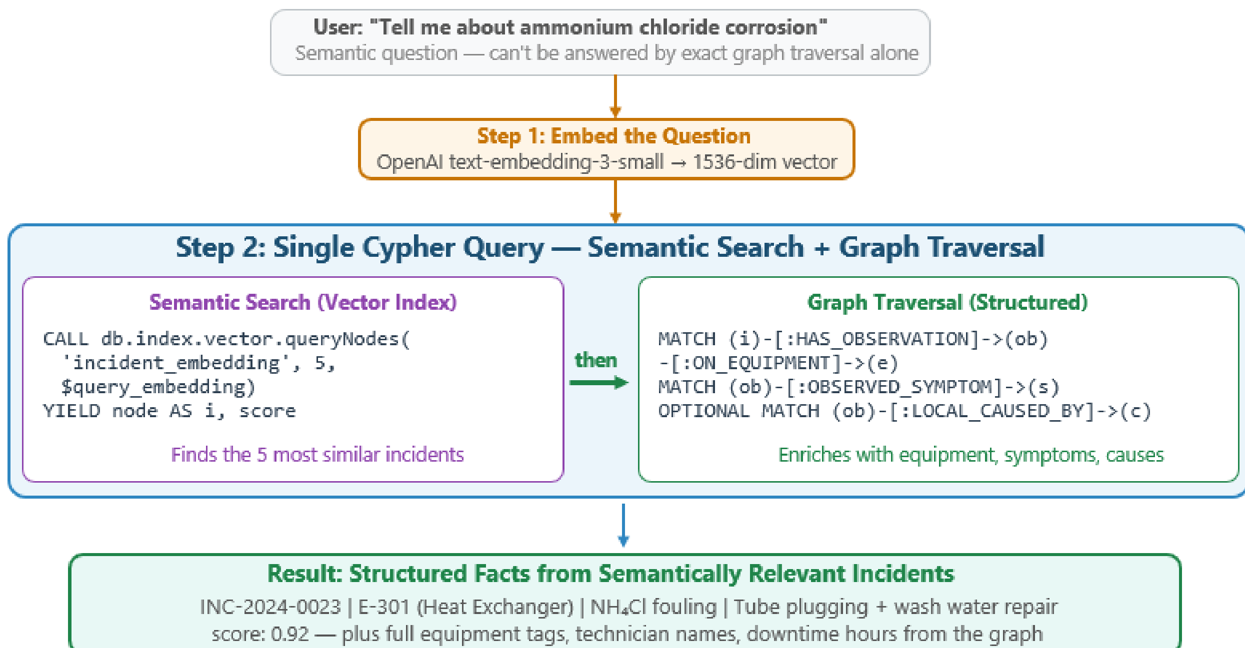
Step: Classify and Execute

```
# Each template has a name, description (for classification), and parameterized Cypher
classification = classify_question(question)  # LLM picks template
# Returns: {'category': 'equipment_history', 'params': {'equipment_tag': 'P-201'}}

template = QUERY_TEMPLATES[classification['category']]
records = session.run(template['cypher'], **classification['params']) # use this data for final answer
                                                                        synthesis via an LLM
```

The picks a template and extracts parameters. The pre-approved Cypher is filled with those parameters and executed. No LLM-generated Cypher touches the database. The trade-off: maximum safety, but questions that don't fit a predefined category get a "not supported" response.

Hybrid Retrieval: Storing Embeddings Inside Your Knowledge Graph

Some questions don't map neatly to graph traversal. "What happened to P-201?" is structural where you traverse from Equipment to Incident. But "Tell me about ammonium chloride corrosion" is semantic; the answer is buried in incident summaries, observation descriptions, and cause narratives. A conventional approach is to maintain a separate vector database alongside Neo4j and stitch results together in application code. But there's a simpler way: store the embeddings directly in the graph as node properties, and use Neo4j's built-in vector index to do semantic search inside Cypher queries.



High-level overview of the steps to implement this hybrid retrieval is discussed below.

Step 1: **Store embeddings during ingestion**. When you ingest an incident, compute an embedding of the summary text and store it as a property on the Incident node:

```
# During ingestion, after creating the Incident node:
embedding = client.embeddings.create(
  model='text-embedding-3-small', input=data['summary']).data[0].embedding

session.run('''
  MATCH (i:Incident {incident_id: id})
  SET i.embedding = embedding''')
```

Each Incident node now carries a 1536-dimensional vector alongside its structured properties (date, severity, downtime). The same approach works for Observation descriptions, Cause names, or any text-heavy property.

Step 2: Create a vector index. Tell Neo4j to index those embeddings for fast similarity search:

```
CREATE VECTOR INDEX incident_embedding
FOR (i:Incident) ON (i.embedding)
OPTIONS {indexConfig: {`vector.dimensions`: 1536, `vector.similarity_function`: 'cosine'}}
```

Step 3: Query with semantic search + graph traversal in one Cypher statement. This is where it gets powerful. A single query finds semantically similar incidents and then traverses the graph to enrich them with structured data:

```
# query_embedding is the embedding for the query
MATCH (i:Incident)
SEARCH i IN (VECTOR INDEX incident_embedding FOR $query_embedding LIMIT 5)
SCORE AS score
MATCH (i)-[:HAS_OBSERVATION]->(ob)-[:ON_EQUIPMENT]->(e)
MATCH (ob)-[:OBSERVED_SYMPTOM]->(s:Symptom)
OPTIONAL MATCH (ob)-[:LOCAL_CAUSED_BY]->(c:Cause)
RETURN i.incident_id, score, e.tag, s.name, c.name, i.downtime_hours
ORDER BY score DESC
```

Read this query from top to bottom: it starts with a vector similarity search (find the 5 incidents whose summaries are most similar to the user's question), then traverses the graph from each result (through Observations to Equipment, Symptoms, and Causes). The output is structured data from semantically relevant incidents; the best of both worlds in a single query.

Summary

In this chapter, you built the GraphRAG query pipeline from the ground up. You connected LangChain to Neo4j, wrote domain-specific Cypher generation and answer synthesis prompts, wired the chain, and tested it against several question types of increasing difficulty. You saw three approaches: LangChain for fast prototyping, raw SDK for full control), and pre-defined templates for production safety (no LLM-generated Cypher).

In Chapter 5, we'll bring everything together into a complete Streamlit web application: a polished interface where engineers can upload reports, build the graph, and ask questions with full Cypher transparency.

Chapter 5

Building a Complete GraphRAG Application

Over the past four chapters, we've built every component of a GraphRAG system: the ontology (Chapter 2), automated extraction (Chapter 3), and multiple query approaches (Chapter 4). Now we bring it all together into a polished web application that a process engineer can actually use. In this chapter, we build a Streamlit-based troubleshooting assistant where engineers can upload incident reports, watch the knowledge graph grow, ask questions in natural language, and see both the answer and the Cypher query that produced it.

Specifically, the following topics are covered:

- Application architecture: how the four layers connect
- The Streamlit app code: session state, caching, upload flow, chat interface
- Cypher transparency: showing engineers the generated query
- When to choose between GraphRAG and simple RAG

5.1 Application Architecture

As shown in Figure 5.1, the application has four layers, each using the best tool for its role. Data flows in two directions. The ingestion path goes: uploaded report → OpenAI extracts entities as Pydantic objects → Neo4j stores them via MERGE. The query path goes: natural language question → LangChain generates Cypher → Neo4j returns records → OpenAI synthesizes a grounded answer. Session state persists chat history and graph status across Streamlit reruns.

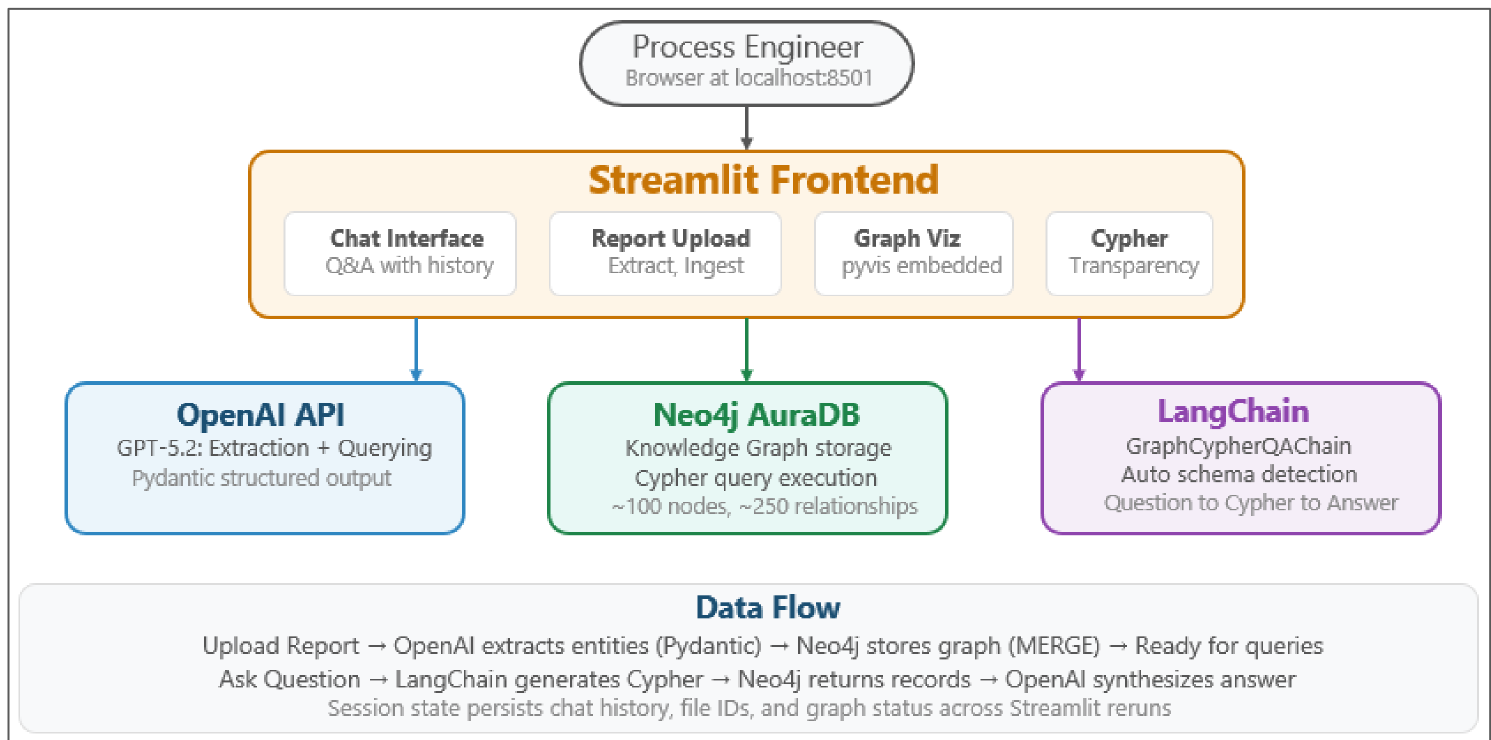


Figure 5.1: Complete Application Architecture

5.2 The Streamlit App: Complete Code Walkthrough

Figure 5.2 shows the web interface that we will build. In this section, we will take a brief look at some of the important aspects of the Streamlit script[14].

[14] If you are not familiar with Streamlit scripting, Book 5 of this series provides a quick refresher.

Figure 5.2: User Interface

Session State Management

Streamlit reruns the entire script on every user interaction. Without session state, variables would be lost between interactions. We initialize three state variables: chat_history stores all previous Q&A pairs (including the generated Cypher), graph_ready tracks whether the graph has data, and ingested_reports prevents re-extracting the same report on every rerun.

```
# These persist across Streamlit reruns
st.session_state.setdefault("chat_history", [])
st.session_state.setdefault("graph_ready", False)
st.session_state.setdefault("ingested_reports", [])
```

Cached Resources

Database connections and LangChain chains are expensive to create. @st.cache_resource ensures they're created once and reused:

```
# see the script app.py for complete code
@st.cache_resource
def get_neo4j_driver():
    return GraphDatabase.driver(
        os.getenv("NEO4J_URI"),
        auth=(os.getenv("NEO4J_USERNAME"), os.getenv("NEO4J_PASSWORD")))

@st.cache_resource
def get_graphrag_chain():
    graph = Neo4jGraph(...)
    chain = GraphCypherQAChain.from_llm(llm, graph, ...)
    return chain, graph
```

When new reports are ingested, we call get_graphrag_chain.clear() to force LangChain to re-read the Neo4j schema, since new node types or properties may have appeared.

The Upload → Extract → Ingest Flow

When a user uploads an incident report, the sidebar processes it in four steps:

- Read the uploaded file as text.
- Call extract_from_report(): our Chapter 3 function that uses OpenAI structured output to extract entities as a typed Pydantic object.
- Call the ingester's ingest_extraction() method: our Chapter 3 MERGE-based ingestion that loads all nodes and relationships into Neo4j.
- Track the ingested file name in session state to prevent re-processing.

```
# Process uploaded file only if it's new (not already ingested)
if uploaded_file and uploaded_file.name not in st.session_state.ingested_reports:
    with st.spinner("Extracting entities with GPT-..."):
        report_text = uploaded_file.read().decode("utf-8")
        extraction = extract_from_report(report_text)
```

```
ingester = KnowledgeGraphIngester()
ingester.setup_constraints()
ingester.ingest_extraction(extraction.model_dump())
ingester.close()

st.session_state.ingested_reports.append(uploaded_file.name)
st.session_state.graph_ready = True
get_graphrag_chain.clear()  # refresh schema
```

The Chat Interface with Cypher Transparency

The chat interface uses Streamlit's *chat_message* and *chat_input* components. The key feature is Cypher transparency: after every answer, an expandable section shows the Cypher query that was generated:

```
# Chat input
question = st.chat_input("Ask a question about your incidents...")

if question:
    with st.chat_message("assistant"):
        result = chain.invoke({"query": question})
        answer = result["result"]
        st.write(answer)
```

Running the Application

You can run your streamlit application coded in file *app.py* via *streamlit run app.py.* The application opens in your browser at http://localhost:8501. You'll see the upload sidebar on the left and the chat interface in the main area. Upload a report (or click "Load Sample Data"), then start asking questions!

When Not to Use GraphRAG!

Reading this book, you would be forgiven for thinking GraphRAG is always better than simple RAG. The wins are real, but they came packaged with costs that are easy to miss because you paid them gradually across five chapters:

- **Ontology design**. Someone (typically you) has to decide what becomes a node, what becomes a property, and how schema changes propagate. This is a real engineering skill that takes effort & time to develop, and is plant-specific.
- **Extraction pipeline**. The LLM extractor, the Pydantic schema, the prompt, the ingester, the entity resolver. Each component has to be tested and maintained. Each extraction costs tokens. Re-extraction after a schema change costs more.
- **Graph operations**. A graph database is another piece of infrastructure: backups, version compatibility, query performance tuning.
- **Cypher generation**. Text-to-Cypher is brittle. Prompt updates require regression testing. Cypher errors at query time produce the worst kind of failure: an answer based on no data, or a database error message shown to an engineer.

A well-tuned simple RAG system has none of these costs. You chunk documents, embed them, retrieve, and synthesize. The whole pipeline fits in 100 lines of Python. When the only thing your users need is to find facts that live inside individual documents, simple RAG is genuinely the better choice.

A decision framework

Use this as a checklist before starting your next GraphRAG project. The more of these statements are true for your situation, the stronger the case for a graph:

Strongly favor GraphRAG when:

- The questions your users actually ask require traversal across multiple documents: cascading failures, repeat patterns, cross-incident aggregation, etc.
- The answers need to cite specific entities (equipment tags, incident IDs) and be auditable. Graph answers are intrinsically structured and easier to ground.
- You expect to keep ingesting new documents over time and want each one to add to a shared, queryable knowledge base, rather than being just another retrievable chunk.

- There is genuine business value in cross-document inferences your users could not previously make. ("Have we seen this combination of symptoms before?")
- Your domain has stable, finite entity types (equipment, incidents, components) that map cleanly onto a graph schema.

Probably prefer simple RAG when:

- The dominant query pattern is "find me the document/section that explains X." SOPs, manuals, regulations, training materials; these are simple RAG territory.
- Documents are largely self-contained: an SOP tells you everything you need to know about starting up the compressor, in one document. Cross-document reasoning is rare.
- The corpus is highly heterogeneous, i.e., every document is shaped differently. Designing an ontology that covers everything well is a poor use of time; vector retrieval is naturally schema-free.
- Your users mostly ask single-hop questions and your simple-RAG baseline already gets them above 80% on your evaluation set.

Summary

With this chapter, we have completed our journey into the world of knowledge graphs and GraphRAG. We started with understanding the need for knowledge graphs for process industry applications and ended with developing a complete GraphRAG application from scratch. With the techniques that you have learnt, viz, ontology design, LLM-powered extraction, graph construction, Cypher querying, and hybrid retrieval, you will be able to build solutions beyond troubleshooting.

End of the book

Of process data science, By process data scientists, For process data scientists

Knowledge Graphs and GraphRAG for Process Industry

This book is designed to help readers quickly gain a working-level knowledge of building knowledge graph and GraphRAG-based applications tailored for process industry operations. The book covers the complete journey from understanding why traditional retrieval-augmented generation falls short on relationship-heavy plant questions, to designing a domain ontology, extracting entities from unstructured incident reports with LLMs, querying the resulting knowledge graph with natural language, and finally deploying everything as a web application. With a hands-on, tutorial-style approach throughout, readers will learn how to set up Neo4j, write Cypher queries that traverse multi-hop relationships, use OpenAI structured outputs for reliable extraction, build a natural-language-to-Cypher pipeline with LangChain, and combine graph traversal with vector search for hybrid retrieval. The application-focused approach of the book is reader-friendly and easily digestible for practicing and aspiring process engineers and data scientists. Upon completion, readers will be able to confidently build and deploy GraphRAG solutions for their plants and make informed design decisions suitable for their industrial environments.

The following topics are broadly covered:

- *Why knowledge graphs and GraphRAG matter for process industry operations*
- *Where traditional approaches (relational databases and simple RAG) fall short*
- *Setting up Neo4j AuraDB and learning the Cypher query language from scratch*
- *Designing a domain ontology for process troubleshooting data*
- *LLM-powered entity and relationship extraction with OpenAI structured outputs*
- *Automated ingestion of incident reports into a Neo4j knowledge graph*
- *Building a natural-language-to-Cypher query pipeline with LangChain*
- *Demo Application: A complete Streamlit-based process troubleshooting assistant*

www.MLforPSE.com