Using Ephemeral Neo4j Instances for a Cybersecurity Risk Graph Scenario

by Dinis Cruz and ChatGPT Deep Research, 2025/06/25

Introduction and Motivation

Graph thinkers – those who naturally conceptualize information as networks of nodes and relationships – can greatly benefit from on-demand graph analytics. Recently, we demonstrated how a Large Language Model (LLM) could serve as an **ephemeral graph database** to help non-programmers build and query a risk management knowledge graph using only natural language. In that approach, the LLM acted as a transient graph engine: nodes and edges existed only in the AI's conversational context (no external database required). This lowers the barrier to entry – no installation or query language needed – but comes with limitations: the "database" resets when the AI session ends, and we rely on the LLM's internal reasoning rather than a true graph query engine.

In this white paper, we bridge that conceptual idea into a practical **Neo4j** implementation. We show how to run the same cybersecurity risk graph scenario on ephemeral (temporary) Neo4j instances. By "ephemeral," we mean spinning up a Neo4j graph database on-demand, using it to perform our analysis, and then tearing it down – much like the LLM's transient memory, but now with a real database. This approach provides the best of both worlds: the full power of a graph database (with Cypher queries, persistence options, and graph algorithms) combined with the flexibility of on-demand usage (no always-on server required). Crucially, it aligns with emerging trends in cloud architecture – treating the database as a disposable, **serverless** component. As described in our technical plan for ephemeral Neo4j on AWS, the idea is to **pay for graph processing only when it runs** and shut everything down when done. Neo4j itself is moving in this direction with offerings like Aura Graph Analytics, which provides **on-demand, ephemeral compute** for graph algorithms.

Use Case: We will walk through a real-world example – modeling a cybersecurity risk management scenario – in a step-by-step, workshop style. This is the same scenario initially built with an LLM, now implemented on Neo4j. The scenario involves a *"User accounts can be compromised"* risk, and expands to include causes (why the risk might happen), impacts (consequences if it happens), controls (mitigations to prevent it), assets (systems affected), and stakeholders (people responsible). By the end, we will have constructed a rich knowledge graph

that connects a technical IT risk to business-level consequences and owners. We'll demonstrate how to create this graph in Neo4j, visualize it, and query it to gain insights. All of this can be done on a temporary Neo4j instance that you spin up just for this analysis – for example, using a Docker container or a Neo4j Aura free tier instance – and discard afterward, achieving true on-demand graph analytics.

Target Audience: This guide is aimed at Neo4j users, engineers, and graph enthusiasts. We assume you have basic familiarity with Neo4j and Cypher, and we focus on the *implementation details* of the ephemeral graph concept. The goal is to empower technical users to replicate the agility of the LLM-based approach within a proper Neo4j environment. By following this workshop, you can run complex graph transformations without maintaining a permanent database server, and integrate the results into your workflows with full provenance and control.

From LLMs as Graphs to Neo4j on Demand

Before diving into the tutorial, it's worth contrasting the LLM-as-graph approach with our Neo4j-based approach:

- **Ephemeral LLM Graphs:** In the LLM approach, the graph lives in the model's memory. Users "create" nodes and relationships by prompting the LLM in plain English (e.g. *"Create a node called User and a node called Account Compromise"*). The LLM interprets and internally represents this structured data. The benefit is a completely natural language interface – no Cypher or setup needed – which makes graph building accessible to non-coders. However, this ephemeral graph is **volatile** and **opaque**. There's no persistent database to query with standard tools, and the fidelity of the graph (and any query results) depends on the LLM's correctness. Complex or large graphs may exceed the context window limits of the model.
- **Ephemeral Neo4j Instances:** In our approach, we spin up a real Neo4j database on demand. This could be done locally (using a Docker container or Neo4j Sandbox for a quick temporary instance) or in the cloud (using an orchestrated ephemeral instance as described in our AWS architecture). The user will use Cypher queries to create nodes and relationships (which requires some technical skill), but gains the **accuracy and power of Neo4j** – including ACID transactions, indexing, Cypher query language, and integration with graph visualization tools. The graph can be saved (if desired) or exported, but by default we treat the database as **throwaway**: once the analysis is done, we extract any results needed and shut the instance down. This means no ongoing resource usage or cost when not in use, analogous to how the LLM's graph vanished after the conversation.

In summary, using an LLM as a graph DB is like scribbling on a whiteboard that an AI holds up – quick and flexible but temporary. Using an ephemeral Neo4j is like booting up a dedicated tool for a task – a bit more effort to start, but you get a reliable engine for as long as you need it, then you

clean up. As we proceed, we'll see how to practically create the same risk graph with Neo4j and discuss the trade-offs and insights gained.

Tutorial: Building a Cybersecurity Risk Graph on an Ephemeral Neo4j Instance

We will now recreate the cybersecurity risk scenario step-by-step in Neo4j. The scenario revolves around the risk statement: **"User accounts can be compromised."** We'll start from this core risk and progressively expand the graph to include impacts, causes, controls, assets, and stakeholders, simulating a realistic threat modeling exercise. For each step, we provide the Cypher queries to input the data and some guidance on visualizing or querying the results in Neo4j.

Ephemeral Setup: To follow along, you can launch a fresh Neo4j instance in a few ways:

- **Locally via Docker:** For example, run `docker run -p 7474:7474 -p 7687:7687 -e NEO4J_AUTH=none neo4j:5` to start a Neo4j database locally with no authentication. This is quick and disposable – when you stop the container, the database is gone (ephemeral storage).
- **Neo4j Sandbox or Aura:** Alternatively, use the Neo4j Sandbox (a temporary cloud instance provided by Neo4j) or Neo4j AuraDB Free. These give you a time-limited database online without installation.
- **Cloud VM/Container:** For a more automated ephemeral setup (as per our AWS architecture), you could script the launch of a Neo4j container on AWS Fargate or an EC2 VM, load data, and destroy it after – but for workshop purposes, a local Docker or Sandbox is simpler.

Once you have Neo4j running, connect to it (e.g., open Neo4j Browser at http://localhost:7474 if using Docker locally). The Neo4j Browser provides a web interface where you can enter Cypher queries, see textual results, and visualize graph data as nodes and relationships.

Note on Data Model: We will use labels to categorize nodes (e.g. `Risk`, `Persona`, `Event`, `Impact`, `Control`, `Cause`, `Asset`, `Role`, `Incident`) corresponding to their roles in the scenario. Relationships will have descriptive types (e.g. `IMPacts`, `INVOLVES`, `LEADS_TO`, `MITIGATES`, etc.). This explicit schema is something we define as we go – similar to how the LLM inferred types on the fly. In Neo4j, you can create nodes and relationships without a predefined schema, which gives us flexibility to mirror the evolving graph thinking process.

Step 1: Initialize an Empty Graph

When using an LLM, we began by "initializing" the graph engine in the conversation (essentially instructing the AI to start with a blank slate memory). In Neo4j, if you just started a fresh instance, you already have an empty database. We should ensure no residual data is present

(especially if reusing a sandbox). You can execute a cleanup query to remove all existing nodes and relationships (skip this on a fresh DB):

```
MATCH (n) DETACH DELETE n;
```

This will detach and delete all nodes in the database, giving us a true blank slate. Neo4j will confirm by returning the number of nodes deleted. We are now ready to start adding our risk data.

Step 2: Creating Nodes (Entities)

Our first task is to capture the basic entities from the risk statement *"User accounts can be compromised."* This statement implies a few key concepts:

- A **Risk** node representing the overall issue (the fact that user accounts can be compromised).
- An **Actor/Persona** node for *User* (who is impacted by the risk).
- An **Event/Outcome** node for *Account Compromise* (the kind of incident we fear).

In the LLM approach, we simply described these nodes in natural language and the AI listed them. Here, we explicitly create them in Neo4j:

```
// Create the core risk node
CREATE (r:Risk {name: "User accounts can be compromised"});
// Create related persona and event nodes
CREATE (u:Persona {name: "User"});
CREATE (e:Event {name: "Account Compromise"});
```

Run these queries (you can execute them one by one or all together). Neo4j will create three nodes. Each node has a `name` property and a label indicating its type. After creation, you might verify the nodes exist:

```
MATCH (n) RETURN labels(n) AS Labels, n.name AS Name;
```

This query fetches all nodes and lists their labels and names. You should see something like:

Labels	Name
`Risk`	User accounts can be compromised
`Persona`	User
`Event`	Account Compromise

At this stage, we have our primary entities, but they are isolated. In the Neo4j Browser's graph view, if you click the small circle icon next to each result row (or run a query returning the nodes), you'll see three separate nodes with no connections. The power of a graph comes from linking these nodes with relationships, which is our next step.

Step 3: Establishing Basic Relationships

Now we will link the nodes to reflect the meaning of the risk statement. From *"User accounts can be compromised"*, we can infer:

- The risk **impacts** the User (if accounts are compromised, users are affected).
- The risk **involves** an Account Compromise event (that's the negative outcome described).

In Neo4j, relationships are directed and have a type. We'll create two relationships:

1. `(:Risk)-[:IMPACTS]->(:Persona)` to show the Risk impacts the User.
2. `(:Risk)-[:INVOLVES]->(:Event)` to show the Risk is about the Account Compromise event.

We can add these by matching existing nodes and creating relationships between them:

```
MATCH (r:Risk {name: "User accounts can be compromised"}),
      (u:Persona {name: "User"})
CREATE (r)-[:IMPACTS]->(u);

MATCH (r:Risk {name: "User accounts can be compromised"}),
      (e:Event {name: "Account Compromise"})
CREATE (r)-[:INVOLVES]->(e);
```

After running these, the database now has two relationships. We can ask Neo4j to show a summary of the graph. For example:

```
MATCH (r:Risk {name: "User accounts can be compromised"})-[]->(x)
RETURN r.name AS Risk, type(last(relationships(path))) AS Relationship, x.name
AS RelatedEntity;
```

(This query finds any relationship from the risk node to another node and returns the type and target's name.) The result should list something like:

- **Risk:** "User accounts can be compromised" — **Relationship:** IMPACTS — **RelatedEntity:** "User"
- **Risk:** "User accounts can be compromised" — **Relationship:** INVOLVES — **RelatedEntity:** "Account Compromise"

In the Neo4j Browser graphical view, if you now visualize the Risk node with its neighbors (e.g. by clicking on the Risk node and expanding relationships), you will see a mini network: the Risk node connected to the User node and the Account Compromise node. This corresponds to the textual

summary above. We've effectively structured the basic risk statement into a graph format. In the LLM-based approach, this was done via conversation; here we did it with a couple of Cypher commands.

This small graph is our starting point. Next, we will expand the graph in two directions: downstream impacts (what happens if the risk materializes) and upstream causes (why the risk might happen), followed by controls and context.

Step 4: Expanding the Graph – Adding Impacts and Consequences

In risk management, a single technical risk can cascade into multiple **impacts**. We will model the consequences if *"User accounts are compromised"*. Based on our scenario, consider these impact points:

- **Sensitive Data Exposed** – if accounts are compromised, sensitive user/customer data could be accessed by an attacker.
- **Breaks GDPR Compliance** – exposure of personal data likely violates regulations (like GDPR).
- **Financial Impact (~ \$500K)** – the data breach and compliance fines could result in an estimated financial loss of, say, \$500K.
- **Company Performance Risk** – ultimately, such a breach can affect the company's performance (e.g., reputational damage, stock impact). We use this as a high-level business risk node.

We will create nodes for each of these and then link them to depict the chain of consequences:

- *Account Compromise* **LEADS_TO** *Sensitive Data Exposed*
- *Sensitive Data Exposed* **CAUSES** *Breaks GDPR Compliance*
- *Sensitive Data Exposed* **RESULTS_IN** *Financial Impact (~ \$500K)*
- *Breaks GDPR Compliance* **RESULTS_IN** *Financial Impact (~ \$500K)* (regulatory fines contribute to the financial loss)
- *Financial Impact (~ \$500K)* **CONTRIBUTES_TO** *Company Performance Risk*

Let's execute this in Cypher. We can do it in batches:

```
// Create impact and consequence nodes
CREATE (i1:Impact {name: "Sensitive Data Exposed"}),
       (c1:ComplianceImpact {name: "Breaks GDPR Compliance"}),
       (b:Impact {name: "Financial Impact (~ $500K)"}),
       (tr:Impact {name: "Company Performance Risk"});

// Create relationships for the consequence chain
MATCH (e:Event {name: "Account Compromise"}),
      (i1:Impact {name: "Sensitive Data Exposed"}),
      (c1:ComplianceImpact {name: "Breaks GDPR Compliance"}),
      (b:Impact {name: "Financial Impact (~ $500K)"}),
```

```
      (tr:Impact {name: "Company Performance Risk"})
CREATE (e)-[:LEADS_TO]->(i1),
       (i1)-[:CAUSES]->(c1),
       (i1)-[:RESULTS_IN]->(b),
       (c1)-[:RESULTS_IN]->(b),
       (b)-[:CONTRIBUTES_TO]->(tr);
```

Here we introduced a new label `ComplianceImpact` for the GDPR-related node, to distinguish it as a compliance issue (you could also just use `Impact`, but giving it a separate label can be useful for clarity). After running these commands, our graph has grown significantly. We can inspect the new subgraph by querying, for example, all impacts stemming from the Account Compromise event:

```
MATCH (e:Event {name: "Account Compromise"})-
[:LEADS_TO|:CAUSES|:RESULTS_IN|:CONTRIBUTES_TO*]->(impact)
RETURN DISTINCT impact.name AS ImpactNode;
```

This query traverses all outgoing impact-related links from *Account Compromise* and should return:

- "Sensitive Data Exposed"
- "Breaks GDPR Compliance"
- "Financial Impact (~ \$500K)"
- "Company Performance Risk"

This confirms our consequence chain. To visualize the hierarchy, you can also step through the chain in the Browser: start from *Account Compromise*, expand the `LEADS_TO` relationship to see *Sensitive Data Exposed*, then expand its outgoing relationships, and so on. You will see a branching structure: Account Compromise → Sensitive Data Exposed → (two branches: Breaks GDPR Compliance and Financial Impact) → and both compliance and financial nodes ultimately connect to Company Performance Risk.

What we've modeled is the reasoning path from a technical event to business risk. As noted in the original LLM-based analysis, this kind of explicit linkage is powerful for explaining *why* a low-level security issue matters in broader terms. In a typical spreadsheet risk register, such relationships might be implicit or lost, but our graph makes them transparent.

Step 5: Incorporating Causes and Preventive Controls

Now that we have the impact side, we turn to the **cause** side – understanding why *"User accounts can be compromised"* in the first place, and what controls exist (or could exist) to prevent it. We'll add contributing factors (causes) to the *Account Compromise* event, and then link mitigation controls to those causes.

Based on our scenario, three primary causes were identified:

1. **Credentials Leaked** – user passwords/credentials are obtained by attackers (e.g., via phishing or a data leak).
2. **No MFA Enabled** – accounts did not have Multi-Factor Authentication, so a stolen password is enough to compromise them.
3. **Threat Not Detected** – the breach wasn't quickly caught by monitoring systems, allowing the attacker to persist.

We model each cause as a node and link them into the graph:

- *Credentials Leaked* **CONTRIBUTES_TO** *Account Compromise*
- *No MFA Enabled* **ALLOWS** *Account Compromise*
- *Threat Not Detected* **AGGRAVATES** *Account Compromise*

Next, for each cause we have an associated preventive control (a policy or measure that mitigates that cause):

- To mitigate **Credentials Leaked**, implement a **Password Policy** (e.g., strong passwords, rotations) and conduct user security training (for simplicity, we'll focus on the policy).
- To mitigate **No MFA Enabled**, enforce an **MFA Policy** (requiring multi-factor auth on accounts).
- To mitigate **Threat Not Detected**, have a **Security Monitoring Procedure** (ensuring security alerts and incident response are in place).

We will create cause nodes and control nodes, then link each control to the cause it mitigates. Additionally, many organizations tie their controls to industry standards or compliance requirements. In our scenario, assume these controls are part of an **ISO 27001 Standard**:

- We add a node for "ISO 27001 Standard".
- We link the standard to each control (relationship like **INCLUDES_CONTROL** or similar) to show these policies are required by the standard.

Let's execute these additions:

```
// Create cause nodes
CREATE (cL:Cause {name: "Credentials Leaked"}),
       (cM:Cause {name: "No MFA Enabled"}),
       (cD:Cause {name: "Threat Not Detected"});

// Link causes to the Account Compromise event
MATCH (e:Event {name: "Account Compromise"}),
      (cL:Cause {name: "Credentials Leaked"}),
      (cM:Cause {name: "No MFA Enabled"}),
      (cD:Cause {name: "Threat Not Detected"})
CREATE (cL)-[:CONTRIBUTES_TO]->(e),
       (cM)-[:ALLOWS]->(e),
       (cD)-[:AGGRAVATES]->(e);

// Create control nodes
```

```
CREATE (p:Control {name: "Password Policy"}),
       (mfa:Control {name: "MFA Policy"}),
       (mon:Control {name: "Security Monitoring Procedure"}),
       (iso:Standard {name: "ISO 27001 Standard"});

// Link controls to causes (mitigations)
MATCH (cL:Cause {name: "Credentials Leaked"}),
      (cM:Cause {name: "No MFA Enabled"}),
      (cD:Cause {name: "Threat Not Detected"}),
      (p:Control {name: "Password Policy"}),
      (mfa:Control {name: "MFA Policy"}),
      (mon:Control {name: "Security Monitoring Procedure"})
CREATE (p)-[:MITIGATES]->(cL),
       (mfa)-[:MITIGATES]->(cM),
       (mon)-[:MITIGATES]->(cD);

// Link standard to controls
MATCH (iso:Standard {name: "ISO 27001 Standard"}),
      (p:Control {name: "Password Policy"}),
      (mfa:Control {name: "MFA Policy"}),
      (mon:Control {name: "Security Monitoring Procedure"})
CREATE (iso)-[:INCLUDES_CONTROL]->(p),
       (iso)-[:INCLUDES_CONTROL]->(mfa),
       (iso)-[:INCLUDES_CONTROL]->(mon);
```

We added three cause nodes, three control nodes, and one standard node, plus the relationships connecting them appropriately. Our graph now has an upstream side (causes) feeding into the risk event, and a downstream side (impacts) leading out of the risk event, with controls mitigating the causes.

To verify, we can query a couple of things:

- List the causes linked to *Account Compromise*:

```
MATCH (cause:Cause)-[r]->(e:Event {name:"Account Compromise"})
RETURN cause.name AS Cause, type(r) AS Relationship;
```

Expected results (order may vary):

- Cause: "Credentials Leaked" – Relationship: CONTRIBUTES_TO
- Cause: "No MFA Enabled" – Relationship: ALLOWS
- Cause: "Threat Not Detected" – Relationship: AGGRAVATES
- List controls mitigating each cause:

```
MATCH (ctrl:Control)-[:MITIGATES]->(cause:Cause)
RETURN ctrl.name AS Control, cause.name AS MitigatedCause;
```

Expected:

- Control: "Password Policy" mitigates "Credentials Leaked"

- Control: "MFA Policy" mitigates "No MFA Enabled"
- Control: "Security Monitoring Procedure" mitigates "Threat Not Detected"
- Check the standard linkage:

```
MATCH (:Standard {name:"ISO 27001 Standard"})-[:INCLUDES_CONTROL]->
(ctrl:Control)
RETURN ctrl.name AS ISO_Control;
```

Expected controls under ISO 27001: "Password Policy", "MFA Policy", "Security Monitoring Procedure".

At this point, the graph is quite comprehensive. If you visualize the whole thing in Neo4j Browser (by doing something like `MATCH (n) RETURN n` and then using the auto-layout), you will see *Account Compromise* roughly in the center:

- To its left/upstream, the cause nodes (Credentials Leaked, No MFA, Threat Not Detected) each connected upward to their mitigating controls and further up to the ISO standard.
- To its right/downstream, the impact nodes (Sensitive Data Exposed, etc.) leading to the Company Performance Risk.
- Above or central, the Risk node “User accounts can be compromised” linking to the Account Compromise event and the User persona.
- We also have the persona “User” hanging off the Risk (with no further connections yet).

This mirrors the diagram we had the LLM generate in the previous approach – except now it’s a live Neo4j graph we can query and update at will. We have one more dimension to add: the human and asset context that grounds this abstract risk in reality.

Step 6: Adding Assets and Stakeholders

So far, our graph is conceptual: it doesn’t yet show which systems and people in our organization are involved. In practice, a risk scenario needs to be tied to actual assets (applications, databases, etc.) and owners (the people responsible for those assets). We will introduce two example systems and their owners:

- **HR System** – an internal system holding employee data. It has good security (strong passwords, MFA enabled).
- **Marketing System** – a system holding customer data for marketing campaigns. It has a weakness: no MFA enforced (just strong passwords).
- **HR Manager** – role/person responsible for the HR System.
- **Marketing Manager** – responsible for the Marketing System.
- **Chief People Officer (CPO)** – the HR Manager reports to this executive.
- **Chief Marketing Officer (CMO)** – the Marketing Manager reports to this executive.

Now, we connect these to our risk graph:

- Both systems are **at risk of** the *Account Compromise* event (since user accounts in those systems could be taken over if credentials leak). We model `(:Asset)-[:AT_RISK_OF]->(:Event)`.
- Each system **implements** certain controls:
- HR System implements Password Policy and MFA Policy.
- Marketing System implements Password Policy but (not) MFA Policy. We can either omit the MFA link to indicate the gap or explicitly represent a lack (for simplicity, we just won't create an MFA implementation for Marketing System, meaning it's missing).
- Ownership relationships:
- HR Manager **OWNS** HR System.
- Marketing Manager **OWNS** Marketing System.
- HR Manager **REPORTS_TO** CPO.
- Marketing Manager **REPORTS_TO** CMO.

Let's add these to the graph:

```
// Create asset and role nodes
CREATE (hr:Asset {name: "HR System"}),
       (mkt:Asset {name: "Marketing System"}),
       (hrMgr:Role {name: "HR Manager"}),
       (mktMgr:Role {name: "Marketing Manager"}),
       (cpo:Role {name: "Chief People Officer"}),
       (cmo:Role {name: "Chief Marketing Officer"});

// Link assets to controls they implement
MATCH (hr:Asset {name: "HR System"}), (mkt:Asset {name: "Marketing System"}),
      (pwd:Control {name: "Password Policy"}), (mfa:Control {name: "MFA
Policy"})
CREATE (hr)-[:IMPLEMENTS]->(pwd),
       (hr)-[:IMPLEMENTS]->(mfa),
       (mkt)-[:IMPLEMENTS]->(pwd);
// (Note: We do NOT create (mkt)-[:IMPLEMENTS]->(mfa) to reflect that Marketing
lacks MFA)

// Link assets to the risk event (at risk of compromise)
MATCH (e:Event {name: "Account Compromise"}),
      (hr:Asset {name: "HR System"}), (mkt:Asset {name: "Marketing System"})
CREATE (hr)-[:AT_RISK_OF]->(e),
       (mkt)-[:AT_RISK_OF]->(e);

// Link ownership of assets
MATCH (hrMgr:Role {name: "HR Manager"}), (mktMgr:Role {name: "Marketing
Manager"}),
      (hr:Asset {name: "HR System"}), (mkt:Asset {name: "Marketing System"})
CREATE (hrMgr)-[:OWNS]->(hr),
       (mktMgr)-[:OWNS]->(mkt);

// Link reporting structure
```

```
MATCH (hrMgr:Role {name: "HR Manager"}), (mktMgr:Role {name: "Marketing
Manager"}),
      (cpo:Role {name: "Chief People Officer"}), (cmo:Role {name: "Chief
Marketing Officer"})
CREATE (hrMgr)-[:REPORTS_TO]->(cpo),
       (mktMgr)-[:REPORTS_TO]->(cmo);
```

Now we have grounded the graph in reality:

- The **HR System** and **Marketing System** nodes attach to the controls and the risk event. Notably, the Marketing System lacks an `IMPLEMENTS MFA Policy` relationship, which is an intentional gap.
- The **User accounts can be compromised** risk now can be contextualized: it affects those two systems (and those systems have different preparedness levels).
- The **stakeholder hierarchy** is captured: each system has an owner, and those owners report to higher management. This will be useful when determining who needs to know about problems.

To check our additions:

- See which systems are at risk of account compromise:

```
MATCH (asset:Asset)-[:AT_RISK_OF]->(:Event {name:"Account Compromise"})
RETURN asset.name AS System, "at risk of Account Compromise" AS RiskRelation;
```

Should list "HR System" and "Marketing System".

- Confirm control implementations:

```
MATCH (asset:Asset)-[:IMPLEMENTS]->(ctrl:Control)
RETURN asset.name AS System, collect(ctrl.name) AS ImplementedControls;
```

Expect:

- HR System implements ["Password Policy","MFA Policy"]
- Marketing System implements ["Password Policy"]

(No MFA in Marketing's list, indicating the gap.)

- Check ownership:

```
MATCH (mgr:Role)-[:OWNS]->(asset:Asset)
RETURN mgr.name AS Owner, asset.name AS Asset;
```

Expect:

- "HR Manager" owns "HR System"
- "Marketing Manager" owns "Marketing System"

- Check reporting:

```
MATCH (mgr:Role)-[:REPORTS_TO]->(exec:Role)
RETURN mgr.name AS Manager, exec.name AS Executive;
```

Expect:

- "HR Manager" reports to "Chief People Officer"
- "Marketing Manager" reports to "Chief Marketing Officer"

Our graph now connects technology, people, and policy. It's evident that the Marketing System is in a riskier state (since it lacks the MFA control), which could be flagged in an analysis. We have effectively set the stage to **simulate an incident**: a credential leak affecting these systems, and then use the graph to deduce consequences and actions.

Step 7: Simulating an Incident and Querying the Graph

Finally, we introduce an incident to test our risk model. Imagine the following scenario: A breach monitoring service reports that a dump of company usernames and hashed passwords has appeared on the internet – a credential leak has occurred (say in June 2025). Among the leaked credentials are accounts from both the HR System and the Marketing System (five accounts from each, for example). This incident can potentially trigger account compromises on those systems.

We will add an **Incident** node to represent this credential leak event, and connect it to our graph:

- *Credential Leak (June 2025)* **AFFECTS** *HR System* (with a detail like "5 accounts" as part of the relationship or node properties).
- *Credential Leak (June 2025)* **AFFECTS** *Marketing System*.
- *Credential Leak (June 2025)* **TRIGGERS** *Account Compromise* (since leaked credentials could lead to accounts being taken over).

Let's add the incident:

```
// Create incident node
CREATE (incident:Incident {name: "Credential Leak (June 2025)"});

// Link incident to affected assets
MATCH (incident:Incident {name:"Credential Leak (June 2025)"}),
      (hr:Asset {name:"HR System"}), (mkt:Asset {name:"Marketing System"})
CREATE (incident)-[:AFFECTS {accounts:5}]->(hr),
       (incident)-[:AFFECTS {accounts:5}]->(mkt);

// Link incident to risk event
MATCH (incident:Incident {name:"Credential Leak (June 2025)"}),
      (e:Event {name:"Account Compromise"})
CREATE (incident)-[:TRIGGERS]->(e);
```

We added a property `{accounts:5}` on the AFFECTS relationships to denote the number of accounts compromised in each system, just as an example of including incident details.

Now, with the incident in place, we can **query the graph to answer important questions**. Unlike the LLM approach where we asked the AI in natural language and it reasoned over the implicit graph， here we can write Cypher queries to get precise answers from the data.

1. Which system is at highest risk now? Given the leak, both systems have some accounts exposed. However, the Marketing System lacks MFA, making it more vulnerable (those leaked passwords can be directly used by attackers). We can find assets that do *not* implement MFA:

```
MATCH (asset:Asset)-[:AT_RISK_OF]->(:Event {name:"Account Compromise"})
OPTIONAL MATCH (asset)-[:IMPLEMENTS]->(:Control {name:"MFA Policy"})
WITH asset, COUNT(*) AS hasMFA
WHERE hasMFA = 0
RETURN asset.name AS AssetWithoutMFA;
```

This will return "Marketing System" as the asset without MFA. In contrast, if we check assets with MFA, HR System would appear. Thus, **Marketing System** is most at risk. (We could also directly see this by noting the missing relationship in the graph or using a Boolean property, but this query demonstrates the approach.)

2. What data could be exposed if the Marketing System is breached? We can follow the chain from the Marketing System to its potential impacts. One way is to traverse from the asset through the risk event to impacts:

```
MATCH (mkt:Asset {name:"Marketing System"})-[:AT_RISK_OF]->(e:Event)-
[:LEADS_TO|:CAUSES|:RESULTS_IN|:CONTRIBUTES_TO*]->(impact)
RETURN DISTINCT impact.name AS Potential_Impact;
```

This will find all impact nodes reachable from *Marketing System* via the Account Compromise event. The results would include "Sensitive Data Exposed", "Breaks GDPR Compliance", "Financial Impact (~ \$500K)", "Company Performance Risk". In other words, if Marketing System accounts are compromised, it could lead to exposure of sensitive customer data, violation of GDPR, a significant financial hit (~\$500K), and ultimately a broader business impact on company performance. This matches the scenario we modeled.

3. Who should be notified about this incident? We want to find the stakeholders responsible for the affected systems. We can traverse the ownership hierarchy from each affected asset:

```
MATCH (incident:Incident {name:"Credential Leak (June 2025)"})-[:AFFECTS]->
(asset:Asset)<-[:OWNS]-(owner:Role)
OPTIONAL MATCH (owner)-[:REPORTS_TO]->(exec:Role)
RETURN asset.name AS AffectedSystem, owner.name AS Owner, exec.name AS
ReportsTo;
```

This query goes from the incident to each asset it affects, finds who owns that asset, and who that owner reports to. The results should be:

- AffectedSystem: HR System – Owner: HR Manager – ReportsTo: Chief People Officer
- AffectedSystem: Marketing System – Owner: Marketing Manager – ReportsTo: Chief Marketing Officer

Thus, the **Marketing Manager and CMO** should be notified for the Marketing System incident, and the **HR Manager and CPO** for the HR System incident. These individuals are accountable for the systems at risk.

4. What immediate actions should be taken? While this question involves some judgment, our graph can highlight actions by pointing to missing controls or active causes:

- The graph shows Marketing System lacks an MFA Policy implementation – an immediate action is *enable MFA on Marketing System* (to close that gap).
- The Password Policy was in place, yet credentials leaked – likely through phishing or reuse. Action: *reset all compromised passwords* (since the graph can't tell us how they leaked, but it shows password policy alone wasn't enough to prevent the leak).
- The presence of *Threat Not Detected* cause suggests if the compromise had happened quietly, it might go unnoticed. If our monitoring didn't catch the misuse of leaked credentials, we need to *improve monitoring and incident response* (the graph node "Threat Not Detected" mitigated by "Monitoring Procedure" hints at this).
- The *Compliance* aspect shows we violated the ISO standard by not having MFA on one system – we should *report this gap and ensure compliance* to avoid audit issues.

We can't get all of these directly via Cypher since some are interpretive, but we can identify that Marketing System is missing an MFA link, which is a red flag:

```
MATCH (mkt:Asset {name:"Marketing System"})
OPTIONAL MATCH (mkt)-[:IMPLEMENTS]->(ctrl:Control {name:"MFA Policy"})
RETURN mkt.name AS System, ctrl IS NULL AS MFA_Missing;
```

This will return "Marketing System" with MFA_Missing = true. So indeed, enabling MFA is an action item. We could also query if any cause nodes are currently not mitigated by controls (though in our graph all causes have a control linked, so that's covered).

By querying and examining the graph, we've effectively performed an **analysis of the incident** using our knowledge graph. In the original LLM session, we asked the AI these questions and it reasoned along the graph; here we wrote explicit queries to derive the answers from the data model. The advantage now is that the results are traceable and we can adjust queries or data as needed.

Step 8: Visualizing the Graph (Optional)

Neo4j Browser will automatically visualize query results that contain nodes and relationships. For instance, if you run a query like `MATCH (n)-[r]->(m) RETURN n,r,m` to fetch a subgraph, the result view shows an interactive node-link diagram. You can also use Neo4j Bloom (a

visualization tool) to create more polished graph visualizations or run Cypher-based search phrases.

For a quick visualization of the **entire graph**, you could run:

```
MATCH (n)-[r]->(m) RETURN n,r,m;
```

This returns all nodes and relationships. In the graph view, you'll see the full picture of our risk scenario. It may be a bit cluttered, so you can manually arrange the nodes:

- Pull the central *Account Compromise* event and *User accounts can be compromised* risk to the center.
- Arrange cause nodes and controls to one side, impact nodes to the opposite side, and assets/roles to another side.
- Alternatively, break it down: visualize upstream causes and controls with one query, and downstream impacts with another. For example: `MATCH (c:Cause)-[*]->(e:Event {name:"Account Compromise"}) RETURN c, e` for the upstream side, and `MATCH (e:Event {name:"Account Compromise"})-[*]->(impact) RETURN e, impact` for downstream.

If needed, you can export the graph or a subgraph. Neo4j supports exporting to formats like GraphML, or you can simply copy the Mermaid diagram format as we did with the LLM (the LLM provided a Mermaid code which we could use to render a diagram). For example, the structure we built could be expressed in Mermaid syntax, or you might use the APOC library (if installed) to export a textual representation. Since this is a hands-on workshop, the Neo4j Browser view might suffice as a visual confirmation.

Conclusion

We have successfully translated the *"LLM as ephemeral graph database"* concept into a tangible Neo4j graph implementation. Using a temporary Neo4j instance, we built a cybersecurity risk knowledge graph that mirrors the one previously created through conversational AI. This exercise demonstrates several key points:

- → **Feasibility of Ephemeral Graph Analytics:** We spun up a Neo4j database on-demand, performed complex graph modeling and querying, and we could just as easily shut it down after extracting insights. This validates the approach proposed in our MVP white paper on ephemeral Neo4j usage – that graph databases can be treated as disposable computation engines, activated only when needed. In a production setting, one could automate this: provision a Neo4j container, load data (e.g., from CSV or S3), run Cypher transformations, output results, then terminate the instance.
- → **Benefits of a Real Graph DB:** Compared to the LLM approach, using Neo4j gave us precise control and reliability. We defined an explicit schema (node labels, relationship types) and could query the graph with well-defined semantics. There's no risk of the AI

"misremembering" a link or hallucinating data – the relationships are exactly as we created them. We also gained the ability to handle larger data (not limited by an AI context window) and to integrate with other tools (for example, connecting the database to BI dashboards or running graph algorithms from Neo4j's GDS library).

- **Interactive Exploration and Queries:** Through Cypher queries, we were able to ask nuanced questions of the graph (e.g., find assets missing a control, traverse impact paths, identify stakeholders) and get answers based on the data. This is akin to how we'd query any Neo4j graph – reinforcing that ephemeral instances don't lose any functionality. You could even run graph algorithms on an ephemeral instance if you needed (for example, PageRank on a subgraph) and then shut it down, illustrating on-demand analytics power.
- **Bridging Non-Technical and Technical Users:** While an LLM interface might be friendlier to non-technical users, this walkthrough shows that with a bit of Cypher, a technical user can achieve the same outcome and more. Importantly, one could wrap this logic behind a simple interface or use an LLM as a front-end to generate Cypher (i.e., the LLM could translate natural language requests into Cypher against an ephemeral Neo4j). This hybrid approach could marry the accessibility of natural language with the robustness of a real database.
- **Ephemerality as a Mindset:** Finally, working with an ephemeral Neo4j encourages good practices like automation and documentation of the build process. Because you don't assume a long-running database, you script the steps to create the graph from scratch. This means the graph's creation becomes **reproducible** and version-controlled (as we have essentially in this document, step by step). If the graph needs to be updated or corrected, you can recreate it. If an analysis needs to be rerun next month with new data, you spin up a fresh instance and execute the script. This ephemeral mindset leads to more reproducible analytics and eliminates configuration drift that often plagues long-lived databases.

In conclusion, empowering graph thinkers in the age of Generative AI can be achieved with a blend of AI and traditional databases. LLMs can inspire new approaches to user interaction with graphs, but Neo4j and other graph databases provide a strong foundation for execution. Using ephemeral Neo4j instances, as we've shown, organizations can perform powerful graph-based reasoning on demand, with cost-efficiency and scalability, without permanently hosting a database. This approach transforms graph analytics into a serverless, flexible service – bring up a graph when you need it, ask your important questions, get your insights, and then gracefully shut it down. The result: more agile decision-making with graphs, and a methodology that keeps pace with the dynamic, ephemeral nature of modern problems and data.