MERIDIAN HEALTH PARTNERS · GOVERNANCE GRAPH

Ontology Manager Guide

How to browse, question, and change the Meridian governance graph: seven tabs, from the schema layer down to individual assets and the actions that mutate them.

BEFORE YOU BEGIN

Getting started

The Ontology Manager is a local web app. It reads and writes a metamodel layer stored inside the same Neo4j database as the Meridian data, so Neo4j must be running before the app starts.

1. Start the Meridian Neo4j instance if it is not already up:
   `NEO4J_CONF=~/neo4j-meridian/conf /opt/homebrew/bin/neo4j start`
2. Seed the metamodel once per database (safe to re-run):
   `NEO4J_URI=bolt://localhost:7688 python3 manager/bootstrap_ontology.py`
3. Start the app server:
   `NEO4J_URI=bolt://localhost:7688 python3 manager/server.py`
4. Open `http://127.0.0.1:8765` in a browser.

The status line under the page title confirms the connection and the shape of the graph, for example: `connected · bolt://localhost:7688 · 199 instance nodes · 440 links · 64 meta nodes`. A red banner means Neo4j is unreachable or the metamodel has not been seeded; the banner says which and what to do.

> **Showcase copy.** The hosted page (ontology_showcase.html) is the same interface running on an embedded snapshot. Everything in this guide applies there too, with one difference: commits are simulated inside your browser and reset when you reload. A banner at the top says so.

TAB 1

Schema browser

The metamodel: what kinds of things exist in the graph (object types), how they may connect (link types), and what each type's properties are. This is where the ontology itself is edited.

a. List

The default view. The left column shows every object type with its color dot, id prefix chip, and live instance count, followed by every link type with its link count. Click any entry to open it in the detail panel on the right.

- **Object type detail** shows the description, id prefix, instance count, whether the type is builtin or custom, and a properties table: name, type, required flag, allowed values for enums, and description.
- **Link type detail** shows the two endpoint types as clickable chips, the direction of the arrow, and a cardinality tag, plus the live link count.
- Both panels offer `Edit` and `Delete`. Deletion is protected: an object type with instances, or one referenced by a link type, refuses to delete and tells you why. A link type with existing links refuses the same way. Nothing in the schema browser can strand data.

b. Map

A force-directed picture of the metamodel: one node per object type in its own color, one labeled arrow per link type. Useful for explaining the model to someone in under a minute.

- **Drag** nodes to arrange the picture; the layout settles around your placement.
- **Hover** a type to see its instance count and property count, and to highlight its links.
- **Double-click** a type to jump to its detail in the List view.
- Self-referencing links such as `COLUMN_OF` and `DERIVED_FROM` draw as small loops on the node.

c. + Object type

Creates a new kind of node. The form asks for:

- **Name**: becomes the Neo4j label. Letters, digits, and underscores, starting with a letter. Names ending in `Def` and metamodel names are rejected so the new type cannot collide with the ontology layer itself.
- **Description**, **id prefix** (the convention new instance ids should follow, such as `DA` or `P`), and a **color** used everywhere the type appears: dots, chips, and both graph views.
- **Properties**: one row per property with a name, a type (`string`, `int`, `float`, `bool`, `date`, `enum`), a required checkbox, and, for enums, the allowed values separated by

commas. Add rows with `+ property`, remove with the row's x.

Save writes the definition into the graph as meta nodes. The new type immediately appears in every tab, with its chosen color, and can be used as a link endpoint.

d. + Link type

Creates a new kind of relationship. The form asks for:

- **Name** in UPPER_SNAKE case, such as `REVIEWED_BY`.
- **From** and **To**: the endpoint object types. The arrow always points from the first to the second.
- **Cardinality**, read along the arrow:
  - `N:1`, each source node holds at most one outgoing link of this type (a person belongs to one org unit).
  - `1:N`, each target node holds at most one incoming link (an acceptance has one owner).
  - `1:1`, both constraints at once.
  - `N:M`, unconstrained.

> **Where cardinality is enforced.** Cardinality is declared here and checked when actions run: a `cardinality` or `noDuplicateLink` validation on an action consults the link type before allowing the write. The schema browser records the rule; the action runner applies it.

TAB 2

Data explorer

Browse and search the actual instances: the 199 nodes and their links.

1. Pick a type from the chips at the top. Each chip shows its live count. The table fills with that type's instances.
2. Type into the search box and press Enter or click `Search`. Matching is against the name and id, case-insensitive. Results page 25 at a time with prev and next.
3. Click a row to open the instance panel: every property, then the node's links grouped by relationship type and direction. `STEWARD_OF ->` lists outgoing links, `<- COLUMN_OF` lists incoming ones.
4. Click any neighbor chip to navigate to that node. A breadcrumb trail at the top of the panel records the path, up to six hops, and each crumb is clickable to backtrack.

This is the fastest way to answer questions like: what feeds this report, who stewards this table, and which columns does this asset have.

TAB 3

Graph view

The whole instance graph as a force-directed picture, read live from Neo4j. Node colors follow the object type colors from the schema browser; table-sized dots are assets, larger dots are hubs with many connections.

- **Hover** a node for a tooltip: type, name, classification or title, and connection count. Neighbors stay lit while everything else dims.
- **Drag** to reposition. `Restart layout` reshuffles from scratch.
- `Show table labels` prints names beside DataAsset tables, the anchors of the picture.
- **Click** a node (a clean click, without dragging) to open it in the Data explorer.
- `Refresh data` re-reads the graph. After you commit an action the view also refreshes itself on your next visit to the tab.

> **Difference from the public demo.** The demo's graph view draws an embedded snapshot. This one queries the live database, so a steward you assigned two minutes ago is already in the picture.

TAB 4

Findings

The tiered HIPAA Security Rule gap analysis, executed live. The tab runs the same two Cypher statements stored in `queries/tiered_gap_analysis.cypher`, so this view, the offline validator, and the paper all share one definition of the scoring.

How a finding is scored. Every PHI or PII asset without a Security Rule control in its compliance chain gets: classification points (PHI 3, PII 2), plus encryption points from its system (absent 2, undocumented 1, verified 0), plus exposure points (a regulatory report downstream 2, any report 1). Score 6 or more is `Tier 1`, 4 to 5 is `Tier 2`, the rest is `Tier 3`.

- The three cards summarize the state: Tier 1 open findings, Tier 2 open findings, and findings suppressed by an active risk acceptance.
- The table lists every finding: tier, asset, classification, system, gap type, status, and score. Status is `open`, `accepted until [date]` in teal, or `acceptance EXPIRED [date]` in red. Expired acceptances resurface in the rollup on purpose.
- `Run analysis` re-executes on demand. The tab also re-runs automatically the next time you open it after committing an action.

> **The loop worth demonstrating.** Note the Tier 1 count. Go to the Action runner, apply a Security Rule control to one of the listed assets, commit, and come back. The finding is gone and the count is lower. Add a risk acceptance instead and the finding shows as suppressed. This is governance changing analysis in real time, and it is the core argument of the project.

TAB 5

Action runner

Where the graph gets changed. An action is a governed mutation: parameterized Cypher plus validation rules, defined in the Action types tab. Nothing is ever written without a preview first.

Running an action, step by step

1. Pick an action from the list, for example **Add risk acceptance**. A form appears, generated from the action's parameter definitions.
2. Fill the parameters. Each kind of parameter gets the right control:
   - 3 **Node parameters** are search pickers. Type part of a name or id, and matching nodes drop down; click one to select it.
   - 4 **Enum parameters** are dropdowns holding the allowed values, pulled from the ontology.
   - 5 **Date parameters** expect `YYYY-MM-DD`.
6. Click `Preview`. The action runs inside a transaction that is then rolled back, so nothing is written. You see either:
   - 7 validation errors in red, one per problem, naming the parameter (an inactive person offered as steward, an id that does not match the required pattern, an expiry date in the past), or
   - 8 a diff card: what would change (`+1 nodes · +2 links · 3 properties set`) and the rows the action returns.
9. If the preview is what you intended, click `Commit`. The action re-validates and runs for real. Commit stays disabled until a clean preview exists, and editing any field disables it again until you re-preview.

After a commit, the graph view and findings tab mark themselves stale and refresh on their next visit, so the effect of the change is immediately visible.

The six builtin actions: assign steward, reclassify asset, add risk acceptance, apply control to asset, deprecate person, and add data quality rule. Deprecate person is worth trying with a preview you then cancel: it lists every asset that would lose its last active steward.

TAB 6

Action types

Where actions are defined. Adding an action type makes a new governed mutation available in the runner.

Adding an action type

1. Click `+ New action type` and give it a name and a one-line description.
2. Write the **Cypher template**. Reference every input as a `$parameter`; values are passed to the driver as parameters and never spliced into the text. End with a `RETURN` naming what changed, so previews read well.
3. Declare the **Params** as a JSON list. Each entry: `name`, `type` (`string`, `int`, `float`, `bool`, `date`, `node`, `enum`), plus `label` for node params, `enumValues` or `enumRef` (such as `"DataAsset.classification"`) for enums, an optional `required` flag, and a `description` the runner shows as a hint.
4. Declare the **Validations** as a JSON list of rules (reference below). Every rule carries a `message` shown to the person running the action.
5. Click `Lint`. Errors block saving; warnings are advisory. Then `Save`.

A complete small example, an action that renames a business term:

```
Cypher:      MATCH (t:BusinessTerm {id: $term_id})
             SET t.name = $new_name
             RETURN t.id AS term, t.name AS now

Params:      [{"name": "term_id", "type": "node", "label": "BusinessTerm",
               "description": "Term to rename."},
              {"name": "new_name", "type": "string",
               "description": "New term name."}]

Validations: [{"rule": "minLength", "param": "new_name", "value": 3,
               "message": "Name needs at least 3 characters."}]
```

Validation rule reference

RULE	FIELDS	CHECKS THAT
`regex`	param, pattern	the value matches the pattern in full, such as `^RA\d{2,3}$`
`minLength`	param, value	the text is at least that many characters

`dateFuture`	param	the date lies after today
`notExists`	param, label	no node of that label already has this id, for id uniqueness
`nodeProp`	param, label, prop, equals	the referenced node's property has the given value, such as Person.active = true
`noDuplicateLink`	from, to, linkType	the link does not already exist between the two chosen nodes
`cardinality`	from, to, linkType	creating the link would not violate the link type's declared cardinality

What the lint blocks

Templates may not contain `DROP`, `CALL dbms`, `CALL apoc`, `LOAD CSV`, constraint or index commands, `FOREACH`, `USE`, multiple statements, unguarded deletes, or any mention of the metamodel labels. Undeclared `$parameters` are errors; unused declared ones are warnings, as is a missing `RETURN`.

> **Builtin actions.** The six seeded actions are marked builtin. You can edit them, but re-running `bootstrap_ontology.py` restores the originals. Copy a builtin into a new action type if you want a lasting variant.

TAB 7

Ask the graph

Plain-language questions, answered from the graph and only from the graph.

Running locally, the flow is the production GraphRAG path: your question is translated into a single Cypher query, a guardrail rejects anything that is not read-only, the query runs against the live database, and the answer is written using only the returned rows. The generated Cypher and the row count are shown above every answer, so each claim can be traced to the query that produced it. If the first query errors, one corrected retry happens automatically. The API key stays on the server; the browser never holds it.

1. Type a question, or click one of the suggestion chips to start from a known-good example.
2. Press Enter or click `Ask`. Generation and querying take a few seconds.
3. Read the trace line first, then the answer. A question the graph cannot answer says so plainly rather than inventing detail.

Questions that work well name things the graph knows: assets, systems, people, reports, regulations. For example: which PHI assets have no steward, what is downstream of ehr.encounters, which acceptances have expired, which assets feeding the CMS report lack data quality rules.

> **In the showcase**, there is no server and no Cypher. Relevant nodes and links are retrieved from the embedded snapshot in your browser and sent as context, and the trace line reports how many. Where model access is not built in, a key box appears: enter an Anthropic API key to ask from your own browser, and tick remember only on a personal machine.
