

Eugene Patent Search — End-to-End Flow

Developer walkthrough: USPTO ETL → Neo4j graph → LLM extraction → FastAPI router → response

Audience

Engineers onboarding to the Eugene biomedical knowledge graph who need a single, file-referenced trace of the patent flow. Every reference uses the form `path/to/file.py:line` so you can open it directly in your IDE and step through with a debugger.

Reference endpoint

`GET /patents/drugs?drug_id=DB00993&page=1&page_size=50` — declared in `src/foundation/router/patent_search_router.py`. The same trace applies to the *clinicaltrials* and *geneproteins* siblings.

How to read this guide

- Sections follow the request path in reverse: start at the HTTP boundary, then walk back through the ETL that fills the graph.
- Section 0 (Schema) is non-obvious and easy to miss: Eugene enforces graph structure by convention, not by Neo4j `CREATE CONSTRAINT`.
- Section 7 contains the verbatim LLM prompts used during patent ingestion.

0. The graph schema (read this first)

Eugene does **not** use Neo4j CREATE CONSTRAINT statements. The schema is conventional, enforced by code:

- **Node labels** are centralized as enums in `src/foundation/model/foundational_node_enum.py` — e.g. `DRUG = 8, "drug", PATENT_APPLICATION = 31, "Patent_Application", CLINICAL_TRIAL = 4, "ClinicalTrial", GENE_PROTEIN = 12, "gene_protein"`.
- **Relationship types** live in `src/foundation/model/foundational_relationship_enum.py` — `DISCLOSED_IN, SUPPORTS_PATENT_APPLICATION, PATENT_APP_TARGET`.
- **Idempotency** comes from MERGE on a natural-key property (e.g. `application_number_text, node_id`) — not from a uniqueness constraint.
- The Neo4j container (`containers/eugene_neo4j_ce/Dockerfile`) ships only `neo4j.conf` + `apoc.conf`; no schema init runs.
- **Seed data** (CSL/Biogen drug assets, drug node properties) lives as plain Cypher in `bin/seed/` and is run manually against a fresh DB.

Implication for debugging: keep the two enum files open while reading any adapter. Every Cypher string interpolates label/relationship strings from them.

1. HTTP entry — the FastAPI router

Wiring

`src/eugene_ws.py:31-78` — `add_routers()` imports `patent_search_router` and includes it in the FastAPI app.

Router file

`src/foundation/router/patent_search_router.py`

Module-load dependency injection (lines 34-35):

```
patent_query_adapter = neo4j_patent_query_adapter()
search_result_mapper = patent_search_result_mapper()
```

Factories are defined in `src/foundation/conf/conf.py:199-206` (adapter + Neo4j driver) and `src/foundation/conf/conf.py:267` (mapper). This is Eugene's manual DI pattern — no framework, no decorators.

Endpoint handler (GET /patents/drugs, lines 38-78)

- Pydantic validation via `Annotated[... , Query(...), AfterValidator(validate_value)]`. Validators in `src/foundation/router/validate_util.py`.
- ID normalization: `normalize_drug_id` in `src/foundation/model/case_helper.py`.
- Adapter call returns a pandas `DataFrame`; mapper turns it into a Pydantic `PatentSearchResult`.

Set your first breakpoint at line 72 (`drug_id = normalize_drug_id(drug_id)`).

2. Query adapter — the Cypher

`src/foundation/infra/db/adapter/neo4j_patent_query_adapter.py`

- `find_related_patents_by_drug` (line 28) opens a session/transaction and delegates to `_find_related_patents_by_drug` (line 107).
- `tx.run(cypher, params).to_df()` → pandas DataFrame.
- Cypher is built in `_generate_patent_search_by_drug` (line 126).
- `ensure_connection()` from `src/graph/util/util.py` is a per-request driver-health guard.
- Pagination math in `src/foundation/infra/db/util/pagination.py`.

Generated Cypher (after label/relationship interpolation)

```
MATCH (n:`drug`)-[r:`disclosed_in`]- (n2:`Patent_Application`)
WHERE n.node_id = $drug_id
RETURN n.node_id AS drug_id,
        n.node_name AS drug_name,
        n2.patent_id AS patent_id
ORDER BY patent_id DESC
SKIP <offset> LIMIT <limit>
```

Response mapping

`src/foundation/mapper/patent_search_result_mapper.py:18-30` iterates the DataFrame and emits `PatentSearchResultRow` instances inside a `PatentSearchResult` (`src/foundation/model/patent/patent_search_result.py`).

That completes the read path. The interesting question is how a `:Patent_Application` node connected by `:disclosed_in` to a `:drug` node ever appears in the graph. That is the ETL.

3. ETL — top-level orchestration

Two CLI entrypoints at the repo root

- `src/download_patents.py` — fetches USPTO applications + pgpub documents to disk as JSON.
- The process path (turns disk JSON into Neo4j nodes/relationships + LLM-extracted knowledge) is invoked from the patent ingest CLIs.

Both paths flow through PatentOrchestrator

`src/patent/provider/patent_orchestrator.py`

- `download()` (line 27) → `ApplicationOrchestrator.fetch_therapeutic_areas` → `JsonWriter.write`, then `PgpubOrchestrator.fetch` → `PgpubJsonWriter.write`.
- `process()` (line 53) → `fetch_and_store_therapeutic_areas` (writes Application nodes) → `PgpubOrchestrator.process(..., with_analysis_and_linking=True)` (writes pgpub nodes + runs the LLM).
- DI wiring: `src/patent/conf/orchestator.py`.

4. ETL — populating Patent_Application nodes

`ApplicationOrchestrator` (`src/patent/application/`) hits the USPTO API (`provider/application_provider.py`) to list recent therapeutic-area applications. Its Neo4j writer adapter MERGES `:Patent_Application` nodes keyed on `application_number_text`. See `src/patent/application/infra/db/`.

5. ETL — populating Pgpub + linking to the Application

`src/patent/pgpub/provider/pgpub_orchestrator.py:55-77` – `process()`

For each Application:

- `pgpub_metadata_provider.list_assoc_documents(...)` → `pgpub_provider.fetch_pgpub(...)` retrieves USPTO pre-grant publications.
- `_add_embeddings(pgpub)` runs the document through `src/patent/pgpub/infra/embedding/pgpub_metadata_embedding_provider.py` and stuffs a vector onto `pgpub.embeddings`.
- `_ingest(pgpub)` → `Neo4jPgpubAdapter.upsert(pgpub)` at `src/patent/pgpub/infra/db/adapter/neo4j_pgpub_adapter.py:32`.

Upsert Cypher (`neo4j_pgpub_adapter.py:50-96`)

```
MERGE (pgpub: `<USPTO_PGPUB_NODE_TYPE>` {
    application_number_text: $application_number_text
})
ON MATCH SET pgpub.refresh_date = datetime(),
            pgpub.abstract = $abstract, ...,
            pgpub.embeddings = $embeddings
ON CREATE SET ... pgpub.is_uspto = true
WITH pgpub
MATCH (application: `<USPTO_APPLICATION_NODE_TYPE>`
    { application_number_text: $application_number_text })
MERGE (application)-[: `<USPTO_PGPUB_PATENT_REL_TYPE>` { is_uspto: true } ]-(pgpub)
```

Label constants live in `src/patent/pgpub/infra/db/const.py` and `src/patent/application/infra/db/const.py`.

This MERGE pattern is the de-facto schema constraint in this codebase: the same `application_number_text` always resolves to the same node, so re-running the ETL is idempotent.

6. ETL — the LLM extraction step (where :disclosed_in is born)

When `with_analysis_and_linking=True`, `PgPubKnowledgeGraphAnalyzer.analyze(...)` runs after ingest. The read-side query in §2 depends on this step, because `(:drug)-[:disclosed_in]-(:Patent_Application)` only exists after the LLM extracts entities from the patent text and the linking step hooks them into canonical drug/gene/disease nodes.

Call chain

- `src/patent/pgpub/analyze/pgpub_knowledge_graph_analyzer.py:22` loops `pgpubs` → `summary_orchestrator.summarize(pgpub)`.
- `src/patent/pgpub/analyze/pgpub_graph_summary_orchestrator.py:64` runs three steps:

Step 1 — extract

`extract_and_write()` (line 88) → `_extract()` (line 177): builds two pages (`pgpub.abstract` and the concatenated `claims`) and runs them through `DocumentAnalyzer.analyze_document_pages`.

Step 2 — link

`upsert_relationships()` (line 109) links extracted entities to the `pgpub` via `Neo4jPgpubLinkingAdapter.link_pgpub_node` — this is what creates the `:disclosed_in`-style edges.

Step 3 — community summaries

`summarize_communities()` (line 129) builds a `NetworkX` graph of extracted entities, runs Leiden-style community detection (`graph/community/`), asks the LLM for a per-community report, and stores summaries as `:Summary/:Finding` nodes via `neo4j_pgpub_summary_adapter.py`.

DocumentAnalyzer

`src/document/analyze/document_analyzer.py:47-85` pairs each `input_param` (which carries the prompt) with each page, fans them out across a `ThreadPoolExecutor` (default 4 workers), streams the LLM response chunk-by-chunk, and returns `PromptResponse`.

Prompt template (`generate.py:476-486`):

```
Question: {question}
Contents to analyze: {document}
```

7. The LLM prompts used during patent ingestion

Wiring: `src/patent/pgpub/conf/graphrag_conf.py:145-149`. The extraction DocumentAnalyzer is constructed with `input_params=[generate_knowledge_graph_extraction_input_params("")]`, which embeds the GraphRAG-style triples prompt below.

7.1 Knowledge graph extraction prompt

[src/infra/llm/prompt/generate.py:205-268](#)

-Goal-

Given a text document, identify all entities and their entity types from the text and all relationships among the identified entities.

-Steps-

1. Identify all entities. For each:

- entity_name (capitalized)
- entity_type
- entity_description

Format: ("entity"/<name>/<type>/<description>)

Only include these entity types:

drug, disease, gene_protein, pathway, anatomy,
biological_process, cellular_component, effect_phenotype,
exposure, molecular_function, chemical_synonym

2. Identify all (source, target) pairs that are clearly related.

- source_entity, target_entity, relation, relationship_description

Format: ("relationship"/<src>/<tgt>/<relation>/<description>)

Only include these relation types:

drug_drug, drug_protein, drug_effect, disease_disease,
disease_protein, pathway_protein, pathway_pathway,
protein_protein, bioprocess_protein, indication

3. Output as JSON with `entities` and `relationships` arrays.

7.2 Entity summary prompt (used by community step)

[src/infra/llm/prompt/generate.py:271](#)

You are a helpful assistant responsible for generating a comprehensive summary of the data provided below. Given one or two entities, and a list of descriptions, all related to the same entity or group of entities, please concatenate all of these into a single, comprehensive description... Make sure it is written in third person, and include the entity names so we have the full context.

Entities: {entity_name}

Description List: {description_list}

7.3 Community report prompt

[src/infra/llm/prompt/generate.py:310](#)

Produces a structured report (TITLE, SUMMARY, IMPACT SEVERITY RATING, RATING EXPLANATION, DETAILED FINDINGS) returned as JSON. The output is parsed by `document/load/stored_summaries_loader.py` and stored as :Summary + :Finding nodes attached to the Patent_Application.

Triples parsing & canonical linking

- `src/graph/mapper/triples_parser.py` parses the ("entity"/...) and ("relationship"/...) lines into Entity/Relationship objects.
- `_link_or_drop_entities` at `pgraph_graph_summary_orchestrator.py:240` calls `Neo4jPgpubLinkingAdapter.find_node_index_by_node_name(entity_type, value)`.
- **This is the canonical-lookup gate:** any LLM-extracted entity that does not already exist as a canonical node (drugs/genes/diseases must be seeded first via `bin/seed/`) is dropped. Surviving entities are linked to the pgraph.

LLM model factory

`src/infra/llm/llm_factory.py` — returns a LangChain `BaseChatModel` (provider/model selected by env vars). Streaming is on by default; the analyzer assembles chunks itself.

8. Suggested debugging walk

- Start FastAPI locally: `uvicorn src.eugene_ws:app`. Then `curl '/patents/drugs?drug_id=DB00993&page=1&page_size=10'`. Breakpoint at `patent_search_router.py:72`. Step into the adapter, watch the Cypher string get built, inspect the DataFrame.
- Once the read path is clear, run `python src/download_patents.py --max-applications 1 --output resources/data` and step through `patent_orchestrator.py:27`.
- Run the `process()` path (find the CLI that calls it — e.g. `convert_patents.py` at the repo root) with a breakpoint at `pgpub_orchestrator.py:55`. Watch `_ingest` write a pgpub node, then enter `PgPubKnowledgeGraphAnalyzer.analyze` to see the LLM extraction.
- Inside `DocumentAnalyzer._analyze_document_task` (`document_analyzer.py:67`), inspect `input_params["question"]` — that is the exact prompt string from `generate.py:205-268` after template interpolation.
- After analyze, query Neo4j: `MATCH (p:Patent_Application)-[:disclosed_in]-(d:drug) RETURN d.node_name, p.patent_id LIMIT 10` — this is the row set the router returns.

9. Reference index (file paths)

Schema

```
src/foundation/model/foundational_node_enum.py
src/foundation/model/foundational_relationship_enum.py
bin/seed/*.cypher
containers/eugene_neo4j_ce/{Dockerfile,conf/}
```

Router & wiring

```
src/eugene_ws.py
src/foundation/router/patent_search_router.py
src/foundation/router/validate_util.py
src/foundation/conf/conf.py
```

Read path (Cypher + mapping)

```
src/foundation/infra/db/adapter/neo4j_patent_query_adapter.py
src/foundation/infra/db/util/pagination.py
src/foundation/mapper/patent_search_result_mapper.py
src/foundation/model/patent/patent_search_result.py
```

ETL orchestrators

```
src/download_patents.py
src/patent/provider/patent_orchestrator.py
src/patent/conf/orchestrator.py
src/patent/application/provider/application_orchestrator.py
src/patent/pgpub/provider/pgpub_orchestrator.py
```

Pgpub upsert & linking

```
src/patent/pgpub/infra/db/adapter/neo4j_pgpub_adapter.py
src/patent/pgpub/infra/db/adapter/neo4j_pgpub_linking_adapter.py
src/patent/pgpub/infra/db/adapter/neo4j_pgpub_summary_adapter.py
src/patent/pgpub/infra/db/const.py
src/patent/application/infra/db/const.py
```

LLM analysis

```
src/patent/pgpub/analyze/pgpub_knowledge_graph_analyzer.py
src/patent/pgpub/analyze/pgpub_graph_summary_orchestrator.py
src/patent/pgpub/conf/graphrag_conf.py
```

```
src/document/analyze/document_analyzer.py
src/infra/llm/prompt/generate.py
src/infra/llm/llm_factory.py
src/graph/mapper/triples_parser.py
src/graph/community/
```