

Eugene TPP Search — End-to-End Flow

Developer walkthrough: pptx ETL → `:cs_l_tpp` + `:cs_l_tpp_question` subgraph → Neo4j vector index (`db.index.vector.queryNodes`) → FastAPI router under `/embeddings` → `SearchResponse`.

Audience

Engineers who need to understand how Eugene matches a **Target Product Profile** (TPP) — a CSL-Behring asset definition slide — against the USPTO patent-application corpus (`:USPTO_PGPUB` nodes) using Neo4j native vector indexes. The four routes on this router are all variations on the same theme: pick an embedding source (free-text, the TPP node, its FastRP graph embedding, or one of its question nodes) and stream the top patent matches through the same vector index. Every reference uses the form `path/to/file.py:line`.

Reference endpoints

- `POST /embeddings/` body `{"include_terms": "Hemophilia A non-viral, in vivo GenM", "exclude_terms": "von Willebrand Disease"}` — ad-hoc free-text search; the include/exclude terms are merged through `SemanticSearchEmbeddingProvider.to_embedding` before hitting the vector index.
- `GET /embeddings/tpps/{tpp_id}` — anchor on an existing `:cs_l_tpp` node and search against the patent text embedding index (`pgpub_embeddings_index`).
- `GET /embeddings/tpps/{tpp_id}/graph` — same anchor, but query the patent **graph** embedding index (`pgpub_prediction_embeddings_index`) using the TPP's own `prediction_embeddings` property (FastRP, see Section 4).
- `GET /embeddings/tpps/{tpp_id}/question/{question_type}` — narrow the anchor to a single question node (`:cs_l_tpp_question`) on the TPP, e.g. `INDICATION` or `MECHANISM_OF_ACTION`, and search with that question's targeted embedding.

Caveat — not registered in `eugene_ws`

At the time of writing, `src/eugene_ws.py` (the FastAPI app factory in `add_routers()`, lines 31-78) **does not** import `tpp.router.search_router`. The router is fully wired internally (module-load DI via `tpp.conf.eugene_ws_conf.tpp_search_orchestrator()`), the orchestrator and adapters are tested, but the router is not included by the core API. Treat this guide as the design doc for the route; before exercising it over HTTP you must add `from tpp.router import search_router` and append it to the `routers` list, or mount it on a separate FastAPI app.

How to read this guide

- Section 0 explains the TPP schema and the two patent vector indexes the router targets.
- Sections 1–3 trace the live request: HTTP entry, query adapter (Cypher), and response mapper.
- Section 4 covers the offline ETL: pptx parsing, embedding, upsert into `:cs_l_tpp` / `:cs_l_tpp_question` and the GDS projection that feeds graph-embedding mode.
- Section 5 is a step-through debugging walk.
- Section 6 is the file-path reference index.

0. Schema (read this first)

The TPP search routes touch two clusters of nodes: the CSL-Behring TPP subgraph (written by the ETL in Section 4) and the USPTO patent-application corpus (:USPTO_PGPUB nodes, ingested by a separate patent pipeline that this guide treats as a given).

```
(:csl_tpp {
  node_id,                # md5(product_description + therapeutc_area + question_types)
  node_index,             # KSUID, set once at creation
  product_description,    # cleaned text from the pptx slide
  threaputic_area,        # TherapeuticAreaEnum.name (usually UNKNOWN)
  embeddings,             # merged sentence-transformer vector
  prediction_embeddings,   # written by FastRP (Section 4)
  refresh_date,
  is_csl = true
})

(:csl_tpp_question {
  node_id,                # md5(question_type + ideal + acceptable + excluded)
  node_index,
  question_type,          # QuestionTypeEnum.name, e.g. INDICATION
  ideal, acceptable, excluded,
  embeddings,             # include-minus-exclude merged vector
  refresh_date,
  is_csl = true
})

(:csl_tpp)-[:has_associated_question { is_csl: true }]-(:csl_tpp_question)

(:USPTO_PGPUB {
  application_number_text, # returned as patent_application_no
  abstract,
  claims,                  # list[str]
  embeddings,              # indexed by pgpub_embeddings_index
  prediction_embeddings     # indexed by pgpub_prediction_embeddings_index
})
```

- **Labels.** CSL_TPP and CSL_TPP_QUESTION are declared in `src/foundation/model/foundational_node_enum.py:42-43` (entries 300 / 301). They are stored on disk as the lowercase strings `csl_tpp` / `csl_tpp_question` via `normalize_node_label()` in `src/tpplib/infra/db/const.py:4-5`.
- **Relationship.** `:has_associated_question` (constant `TPP_QUESTION_REL_TYPE` in `src/tpplib/infra/db/const.py:7`). Carries `is_csl: true` to distinguish CSL assets from any future external TPP imports.
- `tpplib_id == node_id`. The router's path parameter is the md5 hex digest produced by `Neo4jTppAdapter._generate_tpp_composite_node_id()` (`neo4j_tpp_adapter.py:206-223`). FastAPI's `Path(...)` pins it at exactly 32 characters — see Section 1.
- **Two patent vector indexes.** `USPTO_PGPUB_EMBEDDINGS_INDEX_NAME = "pgpub_embeddings_index"` and `USPTO_PGPUB_GRAPH_EMBEDDINGS_INDEX_NAME = "pgpub_prediction_embeddings_index"` (`src/patent/pgpub/infra/db/const.py:9-10`). The first is built from text embeddings on `USPTO_PGPUB.embeddings`; the second from FastRP graph embeddings on `USPTO_PGPUB.prediction_embeddings`.
- **Field-name constants.** `EMBEDDINGS_FIELD_NAME = "embeddings"` and `GRAPH_EMBEDDINGS_FIELD_NAME = "prediction_embeddings"` in

`src/foundation/conf/const.py:4-5`. The adapter interpolates both the index name and the field name into the Cypher, so they must agree across both ends.

- **No Neo4j CONSTRAINTs.** Idempotency relies on MERGE on `node_id` in `Neo4jTppAdapter._generate_tpp_upsert()` (`neo4j_tpp_adapter.py:93-112`). Do not assume database-level uniqueness.

1. HTTP entry — the FastAPI router

`src/tpp/router/search_router.py`

Module-load dependency injection

Lines 26-27 of the router build both the orchestrator and the free-text embedding provider once at import time:

```
search_orchestrator = tpp_search_orchestrator()
embedding_provider = semantic_search_embedding_provider()
```

The factories live in `src/tpp/conf/eugene_ws_conf.py`:

```
def tpp_search_orchestrator() -> TppSearchOrchestrator:
    driver = _neo4j_driver()
    return _tpp_search_orchestrator(
        neo4j_pgpub_search_adapter=_neo4j_pgpub_search_adapter(driver),
        neo4j_tpp_search_adapter=_neo4j_tpp_search_adapter(driver),
    )

def semantic_search_embedding_provider()
    -> SemanticSearchEmbeddingProvider:
    return _semantic_search_embedding_provider(
        embedding_merger=embedding_merger())
```

- One `neo4j.Driver` is shared between `Neo4jTppSearchAdapter` and `Neo4jPgpubSearchAdapter` — the same DI pattern as the rest of Eugene (no FastAPI Depends, no per-request construction).
- The driver comes from `GraphDbConnectionFactory.remote_neo4j_instance_from_env()` (env vars resolved by `.env`).
- `SemanticSearchEmbeddingProvider` is only used by the free-text POST / route; the three GET routes never call it because the anchor TPP node already carries its own embeddings.

Router prefix and the four handlers

```
router = APIRouter(prefix="/embeddings", tags=["embeddings"])

@router.post("/", response_model=SearchResponse,
              response_model_exclude_none=True)
async def find_patents_by_embeddings(
    embedding_search_params: EmbeddingSearchParams):
    merged_embeddings = embedding_provider.to_embedding(
        include_terms=embedding_search_params.include_terms,
        exclude_terms=embedding_search_params.exclude_terms,
    )
    results = search_orchestrator\
        .find_patent_applications_by_embedding(merged_embeddings)
    return to_search_response(results)

@router.get("/tpps/{tpp_id}", ...)
async def find_patents_by_tpp_id(
    tpp_id: Annotated[str, Path(..., max_length=32,
                                min_length=32)]):
    results = search_orchestrator\
        .find_patent_applications_by_tpp(tpp_id=tpp_id)
    return to_search_response(results)

@router.get("/tpps/{tpp_id}/graph", ...)
async def find_patents_by_tpp_id_and_graph_embeddings(...): ...
```

```
@router.get("/tpps/{tpp_id}/question/{question_type}", ...)
async def find_patents_by_tpp_id_and_question_embeddings(
    tpp_id: Annotated[str, Path(..., max_length=32, min_length=32)],
    question_type: Annotated[str, Path(...,
        AfterValidator(validate_question_type))],
):
    question = to_question_enum(question_type)
    results = search_orchestrator\
        .find_patent_applications_by_tpp_question(
            tpp_id=tpp_id, tpp_question=question)
    return to_search_response(results)
```

Path and body validators

- `tpp_id`. All three GET handlers pin `max_length=32`, `min_length=32`. FastAPI returns a 422 automatically for any other length — this is the only sanitisation on the path. Note: no character-class check, but `tpp_id` is bound straight into a Cypher parameter (`$node_id`), not interpolated, so injection is not possible.
- `question_type`. Validated by `validate_question_type` in `src/tpp/router/search_util.py:46-48` which delegates to `to_question_enum()` (lines 31-43). It uppercases the value and looks it up in `QuestionTypeEnum[...]`; an unknown key raises `ValueError`, which FastAPI surfaces as a 422 with the list of valid types in the message:
- `[UNKNOWN, INDICATION, CONTRAINDICATION, MECHANISM_OF_ACTION, ROUTE_OF_ADMINISTRATION, EFFICACY, SAFETY_AND_TOLERABILITY, COST_OF_GOODS_SOLD]` — see `src/tpp/model/question_type_enum.py:4-12`. (Yes, `SAFETY` is the spelling on disk; if you fix it you also fix the data.)
- `EmbeddingSearchParams`. Pydantic model in `src/tpp/router/search_util.py:15-28` with two fields, `include_terms: str` and `exclude_terms: str | None`. No validators — the strings are passed verbatim to the sentence-transformer in `SemanticSearchEmbeddingProvider.to_embedding()`. Unlike the foundational search routes, the TPP router does **not** reuse `validate_value()` from `src/foundation/router/validate_util.py:35-42`.
- **No AfterValidator for the body.** `validate_question_type` is the only `AfterValidator` on the router (line 98 of `search_router.py`). Free-text input bypasses it.

Set your first breakpoint

At `search_router.py:101` (the `question = to_question_enum(question_type)` line) you can inspect the validated enum value before the orchestrator call. For the free-text route, break at `search_router.py:34` to inspect the merged ndarray before it hits the patent vector index.

2. Query adapter — the vector-index Cypher

`src/tpp/provider/tpp_search_orchestrator.py` &
`src/tpp/infra/db/adapter/neo4j_tpp_search_adapter.py`

Orchestrator fan-out

`TppSearchOrchestrator` (lines 16-81) holds two adapters and chooses one per route:

- `find_patent_applications_by_embedding(ndarray) → Neo4jPgpubSearchAdapter.find_by_embeddings(...)` — the only route that goes through the pgpub adapter; the adapter wraps the vector array as a `TppSearchResult(score, application_no)` (no abstract/claims). Mapping happens at lines 41-47.
- `find_patent_applications_by_tpp(tpp_id) → Neo4jTppSearchAdapter.find_by_tpp(tpp_id)` — text vector mode.
- `find_patent_applications_by_tpp_and_graph_embeddings(tpp_id) → Neo4jTppSearchAdapter.find_by_tpp_and_graph_embeddings(tpp_id)` — FastRP graph-embedding mode.
- `find_patent_applications_by_tpp_question(tpp_id, tpp_question) → Neo4jTppSearchAdapter.find_by_tpp_question(...)` — per-question text vector mode.
- None on any required arg short-circuits to None before opening a session — the router surfaces this as an empty `SearchResponse` via `to_search_response(None)`.

Cypher 1 — `find_by_tpp` (text embedding)

`Neo4jTppSearchAdapter._generate_find_by_tpp_id_query()`
(`neo4j_tpp_search_adapter.py:171-181`); `limit=200`:

```
MATCH (tpp:`csl_tpp` { node_id: $node_id })
CALL db.index.vector.queryNodes('pgpub_embeddings_index',
                                200, tpp.embeddings)
YIELD node, score
RETURN score,
       tpp.product_description      AS tpp_description,
       node.application_number_text AS patent_application_no,
       node.abstract                AS patent_abstract,
       node.claims                 AS patent_claims
```

Cypher 2 — `find_by_tpp_question`

`_generate_find_by_tpp_question_query(question_type)`
(`neo4j_tpp_search_adapter.py:183-196`). The `question_type.name.upper()` is spliced directly into the Cypher string — safe because the enum is validated upstream (Section 1):

```
MATCH (tpp:`csl_tpp` { node_id: $node_id })
      -[:has_associated_question]-
      (tpp_question:`csl_tpp_question`
        { question_type: 'INDICATION' })
CALL db.index.vector.queryNodes('pgpub_embeddings_index',
                                200, tpp_question.embeddings)
YIELD node, score
RETURN score,
       tpp.product_description      AS tpp_description,
       node.application_number_text AS patent_application_no,
       node.abstract                AS patent_abstract,
       node.claims                 AS patent_claims
```

Cypher 3 — `find_by_tpp_and_graph_embeddings`

`_generate_find_by_tpp_id_and_graph_embeddings_query()`
(`neo4j_tpp_search_adapter.py:198-210`). Same anchor as Cypher 1, but probes the graph vector index using the TPP's FastRP-derived `prediction_embeddings`:

```
MATCH (tpp:`csl_tpp` { node_id: $node_id })
CALL db.index.vector.queryNodes('pgpub_prediction_embeddings_index',
                                200, tpp.prediction_embeddings)
YIELD node, score
RETURN score,
       tpp.product_description      AS tpp_description,
       node.application_number_text AS patent_application_no,
       node.abstract                AS patent_abstract,
       node.claims                  AS patent_claims;
```

Anatomy

- `db.index.vector.queryNodes(indexName, k, queryVector)`. Neo4j's built-in vector index probe. The index must already exist on `:USPTO_PGPUB` for the matching field; the score is cosine-similarity in `[0, 1]` (configured at index creation time on the patent ingest side).
- **Hard-coded** `limit=200`. Both index queries take the top 200 nearest patents. There is no pagination, no client-controlled `k`, and the `limit` keyword is built into the SQL builder.
- **Parameter usage**. Only `$node_id` is bound (the sanitised 32-char path arg); index names, `k`, and field names are interpolated at query-build time from `USPTO_PGPUB_*_INDEX_NAME / *_FIELD_NAME` constants, so they cannot drift across deployments.
- **Backticked labels**. `:`csl_tpp`` and `:`csl_tpp_question`` use backticks so the lowercase label survives the parser (and matches the on-disk label normalised by `TPP_NODE_TYPE` in `src/tpp/infra/db/const.py:4-5`).
- **Errors**. The transaction body catches `DriverError` and `Neo4jError`, logs the query, and re-raises — FastAPI surfaces them as 500. The router itself has no `try/except`.

Transaction shape

All three TPP-anchored queries flow through `_wrap_tx()`
(`neo4j_tpp_search_adapter.py:212-227`): open a session, open an explicit `begin_transaction()`, run the query callable, return its mapped list. `find_by_tpp_question` reimplements the wrap inline (`neo4j_tpp_search_adapter.py:40-49`) because it needs to pass the `question_type` into the closure.

3. Mapper & response model

`src/tpp/infra/db/model/tpp_search_result.py` & `src/tpp/router/search_util.py`

Row → TppSearchResult

`Neo4jTppSearchAdapter._convert_results()`

(`neo4j_tpp_search_adapter.py:229-247`) is the only mapper for the three TPP-anchored routes:

```
for record in records:
    results.append(TppSearchResult(
        score=float(record["score"]),
        application_no=record["patent_application_no"],
        product_description=record["tpp_description"],
        abstract=record["patent_abstract"],
        claims=record["patent_claims"],
    ))
```

For the free-text POST / route, the orchestrator builds a thinner `TppSearchResult(score, application_no)` inline (`tpp_search_orchestrator.py:41-47`); `product_description`, `abstract` and `claims` are omitted and elided by `response_model_exclude_none=True`.

Pydantic models

```
class TppSearchResult(BaseModel):
    score: float | None = None
    application_no: str | None = None
    product_description: str | None = None
    abstract: str | None = None
    claims: list[str] | None = None
```

```
class SearchResponse(BaseModel):
    count: int
    results: list[TppSearchResult]
```

`to_search_response(results)` (`search_util.py:51-57`) wraps a `None` or empty list as `SearchResponse(count=0, results=[])` — never a 404. Hash & equality on `TppSearchResult` (`tpp_search_result.py:11-22`) key off (`score`, `application_no`) only, so a result set is deduplicable downstream even when the heavyweight fields differ.

Example response — GET /embeddings/tpps/{tpp_id}

```
{
  "count": 3,
  "results": [
    { "score": 0.91,
      "application_no": "US20230012345A1",
      "product_description": "Hemophilia A non-viral, in vivo gene therapy.",
      "abstract": "A method for delivering Factor VIII ...",
      "claims": ["1. A composition comprising ...",
                 "2. The composition of claim 1 ..."] },
    { "score": 0.83,
      "application_no": "US20220098765A1",
      "product_description": "Hemophilia A non-viral, in vivo gene therapy.",
      "abstract": "Lipid nanoparticle formulations encoding ...",
      "claims": ["1. A lipid nanoparticle ..."] },
    { "score": 0.77,
      "application_no": "US20210045678A1",
      "product_description": "Hemophilia A non-viral, in vivo gene therapy.",
      "abstract": "AAV-free delivery of clotting factors ...",
      "claims": ["1. A method comprising ..."] }
  ]
}
```


Example response — POST /embeddings/ (free text)

```
{
  "count": 2,
  "results": [
    { "score": 0.88, "application_no": "US20230012345A1" },
    { "score": 0.74, "application_no": "US20220098765A1" }
  ]
}
```

4. ETL / training pipeline — building the TPP subgraph

`src/tpp/mapper/tpp_mapper.py`, `src/tpp/infra/embedding/tpp_embedding_provider.py`,
`src/tpp/provider/tpp_orchestrator.py`, `src/tpp/infra/db/adapters/neo4j_tpp_adapter.py`,
`src/tpp/provider/similarity_search_train_orchestrator.py`

The live router reads two properties that are written entirely offline: `cs_l_tpp.embeddings / cs_l_tpp_question.embeddings` (sentence-transformer vectors), and `cs_l_tpp.prediction_embeddings` (FastRP graph embeddings). Both flow through this pipeline.

Step 1 — pptx parsing

- `TppMapper.map(pptx)` (`src/tpp/mapper/tpp_mapper.py:19-25`) opens a PowerPoint deck and walks every slide. A slide is recognised as a TPP when any shape's text starts with "Target Product Profile" (`tpp_mapper.py:50-52`).
- For each TPP slide, the mapper concatenates the slide text into `product_description` (cleaned by `_clean_tpp_product_description()` which strips "Confidential – Internal Use Only", pipes, and the "Target Product Profile" marker), and walks every table looking for the question grid.
- `_map_row_to_question(row)` reads four columns — *type*, *ideal*, *acceptable*, *excluded* — and maps the first column through `_map_col_to_question_type()` into a `QuestionTypeEnum`. Rows whose header is not one of the seven known labels are dropped silently.
- Output is a `set[Tpp]` (one Tpp per slide, deduped by hash of the composite key).

Step 2 — embeddings

`TppEmbeddingProvider.apply_embeddings(tpp)` (`tpp_embedding_provider.py:19-32`) is called for every TPP and sets the `embeddings: ndarray` field on both the TPP and each of its questions.

- **Base TPP embedding.** `_to_base_embeddings()` (`tpp_embedding_provider.py:54-67`) encodes `product_description` and `therapeutic_area.name` via `self.model.encode(chunks, show_progress_bar=False)`.
- **Per-question embedding.** `_question_to_embedding()` (`tpp_embedding_provider.py:69-79`) builds *positive* chunks from (`question_type.name`, `acceptable`, `ideal`) and a *negative* chunk from `excluded`, then calls `EmbeddingMerger.merge(embeddings=[pos], exclusion_embeddings=[neg])` to obtain an include-minus-exclude vector. Same merger is reused for the free-text route — `SemanticSearchEmbeddingProvider.to_embedding(include_terms, exclude_terms)`.
- **Final TPP embedding.** `to_embeddings()` (`tpp_embedding_provider.py:34-52`) concatenates the base chunks with every question's positive chunks and calls the merger one last time to produce the single `ndarray` stored on `cs_l_tpp.embeddings`.

Step 3 — upsert

`TppOrchestrator.process(tpp_file)` (`tpp_orchestrator.py:49-73`) is the entry point. It calls the mapper, applies embeddings, then ingests each TPP via `Neo4jTppAdapter.upsert(tpp)` (`neo4j_tpp_adapter.py:35-47`). Composite `node_id` values are generated by `_generate_tpp_composite_node_id()` (md5 over the product description, therapeutic area, and the sorted question type names) and `_generate_tpp_question_composite_node_id()` (md5 over `question_type + ideal + acceptable + excluded`). `node_index` is a KSUID set once at creation.

Upsert Cypher — Neo4jTppAdapter._generate_tpp_upsert()
(neo4j_tpp_adapter.py:93-112):

```
MERGE (tpp:`csl_tpp` { node_id: $node_id })
ON MATCH
    SET tpp.threaputic_area      = $threaputic_area,
        tpp.product_description = $product_description,
        tpp.refresh_date        = datetime(),
        tpp.embeddings          = $embeddings
ON CREATE
    SET tpp.threaputic_area      = $threaputic_area,
        tpp.product_description = $product_description,
        tpp.refresh_date        = datetime(),
        tpp.node_index          = $node_index,
        tpp.embeddings          = $embeddings,
        tpp.is_csl              = true
```

Per-question MERGE blocks (neo4j_tpp_adapter.py:140-176) chain via WITH tpp and connect with MERGE (tpp)-[r1:`has\_associated\_question` { is_csl: true }]- (tpp_question). All chunks are joined into one Cypher transaction terminated by RETURN tpp.node_id, tpp.node_index, tpp.refresh_date.

Step 4 — graph projection for FastRP

The GET /embeddings/tpps/{tpp_id}/graph route depends on csl_tpp.prediction_embeddings being populated by FastRP over a GDS in-memory projection. SimilaritySearchTrainOrchestrator.project_for_similarity_search() (src/tpp/provider/similarity_search_train_orchestrator.py:29-33) projects the SIMILARITY_GRAPH_PROJECTION_NAME graph with the following labels and relationships, then a sibling FastRP training step writes prediction_embeddings back onto each node (the same pipeline used by /similarity/{label} — see the Similarity flow guide for the FastRP Cypher):

```
source_node_labels = [DISEASE, DRUG]
target_node_labels = [ANATOMY, DISEASE, DRUG, EFFECT_PHENOTYPE,
                      EXPOSURE, GENE_PROTEIN, PATHWAY]
relationship_labels = [CONTRAINDICATION,
                      DISEASE_PHENOTYPE_NEGATIVE,
                      DISEASE_PHENOTYPE_POSITIVE,
                      DISEASE_PROTEIN, EXPOSURE_DISEASE,
                      INDICATION, OFF_LABEL_USE,
                      PATHWAY_PATHWAY, PATHWAY_PROTEIN]
self.neo4j_project_similarity_graph_adapter\
    .project_similarity_graph(
        projection_graph_name=SIMILARITY_GRAPH_PROJECTION_NAME,
        source_node_labels=source_node_labels,
        target_node_labels=target_node_labels,
        relationship_labels=relationship_labels,
        replace=True,
    )
```

- Note that CSL_TPP is **not** in the source/target label list — the projection only spans the foundational medical graph. The TPP's own prediction_embeddings is therefore expected to be written by a separate FastRP run that explicitly includes :csl_tpp; verify this in your deployment before relying on the graph route.
- The patent ingest pipeline (out of scope here) creates pgpub_embeddings_index and pgpub_prediction_embeddings_index over :USPTO_PGPUB. Without those indexes, db.index.vector.queryNodes(...) in Section 2 raises “there is no such vector schema index”.

5. Suggested debugging walk

- **Step 1 — bring up the stack.** `docker-compose up eugene_neo4j eugene_ws`. Confirm that both patent vector indexes exist: `SHOW VECTOR INDEXES YIELD name` in Neo4j Browser should list `pgpub_embeddings_index` and `pgpub_prediction_embeddings_index`. If the router is not registered (see the Caveat in the cover), temporarily include it in `add_routers()` of `src/eugene_ws.py`.
- **Step 2 — load TPPs.** Drop a CSL pptx into the watched ingest folder and run the TPP orchestrator entry point. Breakpoint at `tpp_orchestrator.py:61` to watch `TppMapper.map` parse one slide, then at `tpp_orchestrator.py:66` to inspect the embedded ndarrays before `Neo4jTppAdapter.upsert` runs.
- **Step 3 — pick a tpp_id.** In Neo4j Browser: `MATCH (t:csl_tpp { is_csl: true }) RETURN t.node_id, t.product_description LIMIT 5`. Copy a 32-char hex id.
- **Step 4 — hit the routes.** `curl -sS http://localhost:8000/embeddings/tpps/<tpp_id> | jq .` for the text-vector path. Add `/graph` for the FastRP path, or `/question/INDICATION` for the per-question path. For free text: `curl -sS -X POST http://localhost:8000/embeddings/ -H 'Content-Type: application/json' -d '{"include_terms": "Hemophilia A non-viral, in vivo GenM", "exclude_terms": "von Willebrand Disease"}' | jq ..`
- **Step 5 — breakpoint sweep.** Set breakpoints at `search_router.py:56` (right before the orchestrator call — inspect the parsed path arg), `tpp_search_orchestrator.py:57` (delegation into the adapter), and `neo4j_tpp_search_adapter.py:81` (the actual tx.run). Copy the assembled Cypher into Neo4j Browser with `{node_id: '...'}` to see the raw 200-row index result before mapping.
- **Step 6 — force the failure modes.** Send a 31-char id → FastAPI 422 from the Path constraint. Send an unknown question type like `/question/banana` → 422 from `validate_question_type` with the list of valid values. Send an id that does not exist → MATCH returns zero anchors, the adapter returns an empty list, and `to_search_response()` emits `{count: 0, results: []}` (200, not 404).

6. Reference index

HTTP entry	
src/tpp/router/search_router.py	# router + DI L26-27, handlers L30-106
src/tpp/router/search_util.py	# SearchResponse, EmbeddingSearchParams, # to_question_enum, validate_question_type, # to_search_response
src/foundation/router/validate_util.py	# validate_value (not used here, reference only)
DI / wiring	
src/tpp/conf/eugene_ws_conf.py	# tpp_search_orchestrator(), # semantic_search_embedding_provider()
src/eugene_ws.py	# NB: search_router NOT yet registered
Orchestration	
src/tpp/provider/tpp_search_orchestrator.py	# TppSearchOrchestrator L16-81
Query adapter (live request path)	
src/tpp/infra/db/adapter/ neo4j_tpp_search_adapter.py	# _generate_find_by_tpp_id_query L171-181 # _generate_find_by_tpp_question_query L183-196 # _generate_find_by_tpp_id_and_graph... L198-210 # _wrap_tx L212-227 # _convert_results L229-247
src/patent/pgpub/infra/db/adapter/ neo4j_pgpub_search_adapter.py	# find_by_embeddings (free-text route)
src/patent/pgpub/infra/db/const.py	# USPTO_PGPUB_EMBEDDINGS_INDEX_NAME L9-10
src/foundation/conf/const.py	# EMBEDDINGS_FIELD_NAME L4-5
Response	
src/tpp/infra/db/model/tpp_search_result.py	# TppSearchResult
src/tpp/router/search_util.py	# SearchResponse, to_search_response
ETL / training pipeline	
src/tpp/mapper/tpp_mapper.py	# pptx -> set[Tpp]
src/tpp/infra/embedding/tpp_embedding_provider.py	# sentence-transformer + EmbeddingMerger
src/tpp/provider/tpp_orchestrator.py	# process(), _apply_embeddings, _ingest
src/tpp/infra/db/adapter/neo4j_tpp_adapter.py	# upsert L35-91, MERGE template L93-112, # question MERGE L114-178, composite id L206-239
src/tpp/infra/db/const.py	# TPP_NODE_TYPE, TPP_QUESTION_NODE_TYPE, # TPP_QUESTION_REL_TYPE
src/tpp/provider/similarity_search_train_orchestrator.py	# GDS projection for FastRP
Schema enums	
src/foundation/model/foundational_node_enum.py	# CSL_TPP (300), CSL_TPP_QUESTION (301) # USPTO_PGPUB (201)
src/tpp/model/question_type_enum.py	# INDICATION, CONTRAINDICATION, # MECHANISM_OF_ACTION, ROUTE_OF_ADMINISTRATION, # EFFICACY, SAFETY_AND_TOLERABILITY, # COST_OF_GOODS_SOLD
src/tpp/model/therapeutic_area_enum.py	# TherapeuticAreaEnum (UNKNOWN today)
src/tpp/model/tpp.py	# Tpp domain object
src/tpp/model/tpp_question.py	# TppQuestion domain object