MetaKG: A Unified Knowledge Graph System for Metabolic Pathway Analysis

Eric G. Suchanek, PhD[1]

[1]Flux Frontiers, github.com/Flux-Frontiers

February 2026

Abstract

A fundamental challenge in systems biology is that metabolic pathway data is fragmented across incompatible formats and disconnected from the semantic and structural query capabilities required by modern computational workflows. We present MetaKG, a local-first knowledge graph system that unifies metabolic pathway data from four formats (KGML, SBML, BioPAX, CSV) into a unified property graph with a novel dual-layer architecture: SQLite for efficient structural graph traversal and LanceDB for semantic similarity search via dense vector embeddings. The core innovation is that MetaKG combines four orthogonal query modalities—structural neighbourhood traversal, breadth-first shortest-path search, natural-language semantic similarity, and full stoichiometric detail assembly—all through a unified Python API, CLI, and Model Context Protocol server interface. The system requires no external services or database servers; queries run entirely on local machine after building a snapshot graph from input files. A stable URI-style identifier scheme enables deterministic cross-format merging, making incremental builds tractable and the graph reproducible. We demonstrate that this architecture enables metabolic pathway analysis workflows that are simultaneously precise (structural), exploratory (semantic), and AI-accessible (MCP integration). The system is evaluated on the complete human metabolome: all 369 KEGG pathways (22,290 nodes, 11,298 edges), integrating metabolic, signaling, and regulatory networks. Performance analysis shows that the dual-layer architecture scales efficiently to organism-scale pathway corpora, with typical queries completing in milliseconds to seconds.

Keywords: metabolic pathways, knowledge graph, dual-layer architecture, semantic search, local-first systems, AI-accessible data, multi-format integration, bioinformatics

1 Introduction

1.1 The Problem: Query Architecture Limits Exploratory Analysis

Reconstructing metabolic networks requires integrating diverse data sources: KEGG provides human-curated pathway maps in KGML [Kanehisa et al., 2021]; Reactome and model-organism databases export in BioPAX Level 3 [Demir et al., 2010]; constraint-based modelling tools produce SBML [Keating et al., 2020]; and institutional data often exists only in spreadsheets. While handling multiple formats is a technical necessity, it is not the core scientific problem. The deeper issue is query capability.

Existing systems force a false choice between two incompatible query paradigms:

Structural precision Systems like KEGG, BioCyc, and Reactome offer exact structural queries ("retrieve all products of this enzyme", "shortest path between two compounds") through relational data models or graph databases. But they lack semantic search; a biologist searching for "fatty-acid oxidation" gets keyword matches, not conceptually similar pathways. They also require web access and cannot be queried programmatically at scale.

Semantic expressivity Vector databases and embeddings excel at semantic similarity ("find pathways similar to X"), handling synonymy and domain terminology. But they abandon the property graph structure; stoichiometric detail, cofactor roles, and regulatory logic are lost. Semantic-only systems are exploratory tools, not precision instruments.

No existing system combines structural precision, semantic expressivity, and programmatic accessibility in a single local-first tool. KEGG and PathBank are web-only. MetaNetX reconciles identifiers but provides no queryable graph. Neo4j-based systems support complex structural queries but require server deployment and do not address parsing. Vector databases excel at search but discard relational structure. In short, no unified platform enables an analyst to start with a semantic query ("what pathways are related to glucose metabolism?"), drill into structural detail ("what is the precise stoichiometry of this reaction?"), and assemble the results all within a single local, reproducible, programmatically-accessible system.

1.2 The Solution: Dual-Layer Query Architecture

MetaKG solves this dilemma with a dual-layer local knowledge graph that breaks the false choice between structural precision and semantic expressivity. The core innovation is separating query problems by modality:

Structural layer SQLite stores the property graph (compounds, reactions, pathways, and their relationships). Structural queries run as efficient SQL joins: neighbourhood traversal, shortest-path BFS, stoichiometric assembly.

Semantic layer LanceDB maintains a vector index over node descriptions using sentence-transformer embeddings. Semantic queries retrieve results by cosine similarity: natural-language pathway search, compound similarity, synonym handling.

This architecture enables analysts to use four query modalities on the same graph without choosing a single paradigm. Start with semantic exploration, then drill into structural detail, all in one interface:

1. **Semantic pathway discovery** — "Find pathways related to fatty-acid beta-oxidation" (vector similarity search; handles synonyms, abbreviations, domain terminology)
2. **Structural neighbourhood traversal** — "Find all compounds that are products of pyruvate carboxylase" (SQL joins)
3. **Shortest-path search** — "What is the minimal metabolic route from glucose to acetyl-CoA?" (BFS over compound–reaction bipartite graph)
4. **Stoichiometric detail assembly** — "Retrieve all substrates, products, enzymes, inhibitors, and cofactors for reaction R00200" (multi-table JOIN with JSON unpacking)

Multi-format parsing (KGML, SBML, BioPAX, CSV) is the enabling infrastructure. It allows users to ingest data from any source and merge it by stable, deterministic identifiers. But parsing is table stakes. The innovation is the dual-layer query engine: it frees users from choosing between semantic expressivity and structural precision.

All four query modalities are exposed through a unified Python API, command-line interface, and Model Context Protocol (MCP) server. The system runs entirely locally: no network, no database server, no external services after setup. This is a deliberate design choice. Unlike live database mirrors, MetaKG treats the knowledge graph as a reproducible, version-controlled snapshot. Users rebuild when input files change. This enables reproducible analysis, offline workflows, and integration into research reproducibility pipelines. The system scales efficiently to organism-scale pathway corpora: the complete *Homo sapiens* metabolome (369 KEGG pathways, 22,290 nodes, 11,298 edges) is built in 30–60 seconds and queries complete in milliseconds to seconds.

1.3 Design Goals

To realise this vision, MetaKG is built around three core principles, in priority order:

1. **Dual-layer query architecture.** Combine structural and semantic queries in a single unified system. Do not force users to choose between precise graph traversal and exploratory semantic search; both modalities should work seamlessly on the same data through the same API.
2. **Format agnostic, deterministic merging.** Accept pathway data in any of four formats (KGML, SBML, BioPAX, CSV) and produce a unified graph using stable, deterministic identifiers. Enable reproducible builds and incremental updates without external reconciliation services.
3. **Zero-friction local deployment.** Require no database servers, external services, or network connections after setup. The entire stack (parsing, storage, indexing, querying, visualisation, MCP server) fits in a single Python library, runnable on a laptop or integrated into batch workflows.

The remainder of this paper is organised as follows. Section 2 describes the data model and system architecture. Section 3 covers the format-specific parsers. Section 4 details the storage and indexing layers. Section 5 describes the query API. Section 6 covers the visualisation components. Section 7 describes the MCP server. Section 8 walks through a complete worked example. Section 10 discusses limitations and future directions, and Section 11 concludes.

2 Data Model and Architecture

2.1 Property Graph Schema

MetaKG represents metabolism as a directed property graph $G = (V, E)$. Vertices V are *entities* of four kinds:

`compound` A small molecule metabolite. Carries optional molecular formula, net formal charge, and cross-references to external databases (ChEBI, HMDB, PubChem, InChI).

`reaction` A biochemical transformation. Carries stoichiometry encoded as a JSON object listing substrates and products with their coefficients, a reversibility flag, and cross-references to KEGG and Rhea.

`enzyme` A protein catalyst. Carries the EC number and cross-references to UniProt and NCBI Gene.

`pathway` An ordered or thematic collection of reactions, typically corresponding to one named pathway in a source database.

Edges E carry one of seven relation types (Table 1). All edges are directed and carry an optional evidence blob encoded as JSON, which parsers use to record stoichiometric coefficients, compartment labels, and source-specific annotations.

Table 1: Edge relation types in the MetaKG graph schema.

Relation	Source kind	Target kind
`SUBSTRATE_OF`	compound	reaction
`PRODUCT_OF`	reaction	compound
`CATALYZES`	enzyme	reaction
`INHIBITS`	compound	reaction
`ACTIVATES`	compound	reaction
`CONTAINS`	pathway	reaction
`XREF`	any	any

2.2 Stable Identifier Scheme

Identifier reconciliation is one of the core difficulties in biological data integration. MetaKG assigns each node a stable, URI-style string identifier of the form:

```
<prefix>:<namespace>:<external-id>
```

where the prefix encodes the node kind (`cpd`, `rxn`, `enz`, `pwy`) and the namespace identifies the source database (`kegg`, `chebi`, `uniprot`, `ec`, etc.):

```
cpd:kegg:C00022        # Pyruvate (KEGG)
rxn:kegg:R00200        # Pyruvate decarboxylation
enz:ec:1.2.4.1         # Pyruvate dehydrogenase (EC)
pwy:kegg:hsa00010      # Glycolysis / Gluconeogenesis
```

Listing 1: Examples of stable node identifiers.

For entities that appear in a file without an external database identifier, a synthetic identifier is constructed by hashing the lowercased display name with SHA-1 and retaining the first eight hexadecimal digits:

`cpd:syn:a4f2b8c1`

Because the hash is applied to the normalised name, identifiers are deterministic across independent parser runs on the same input, enabling incremental rebuilds without producing duplicate nodes. When two source files refer to the same KEGG or ChEBI entry, the graph merges their nodes by ID before writing to SQLite, so each logical entity appears exactly once in the graph regardless of how many files contributed to it.

2.3 System Architecture and the Dual-Layer Query Engine

Figure 1 shows the overall pipeline. The `MetaKG` orchestrator class owns the full pipeline from raw files to query results. Internally it coordinates three subsystems:

1. **MetabolicGraph** — responsible for file discovery and parser dispatch. Outputs normalised `MetaNode` and `MetaEdge` objects that are independent of source format.

2. **MetaStore** — SQLite persistence layer that enables structural graph queries (neighbourhood traversal, shortest-path BFS, stoichiometric assembly) via efficient SQL joins. Provides ACID guarantees and requires no external services.

3. **MetaIndex** — LanceDB vector index layer that enables semantic queries (natural-language pathway search, compound similarity retrieval). Uses pre-trained sentence-transformer embeddings to handle synonymy, abbreviations, and domain terminology.

The dual-layer design is the core architectural innovation. By separating structural queries (which are naturally SQL problems) from semantic queries (which are naturally vector problems), MetaKG avoids the false choice between relational precision and semantic expressivity. A user can traverse the graph via stoichiometry using SQL, then search for semantically similar pathways using embeddings, all within the same interface.

Both the store and the index are initialised lazily; a caller that builds only the SQLite database (e.g. with `-no-index`) never instantiates the embedding model. This enables lightweight deployments where only structural queries are needed, and avoids the 100 MB embedding model download unless semantic search is actually used.

Listing 2: High-level data flow.

```
Pathway files (KGML / SBML / BioPAX / CSV)
        |
  MetabolicGraph
  (file discovery + parser dispatch)
        |
  MetaNode / MetaEdge objects
  (merged by stable ID)
        |
  MetaStore  ----> SQLite
  (write + xref index)
        |
  MetaIndex  ----> LanceDB
  (sentence-transformer embeddings)
        |
  Query API + CLI + MCP server
```

Figure 1: MetaKG data pipeline. Format-specific parsers produce a stream of normalised `MetaNode` and `MetaEdge` objects that are merged by stable identifier, persisted to SQLite, and indexed as dense vectors in LanceDB.

3 Format-Specific Parsers

MetaKG ingests metabolic pathway data from four standard formats: KEGG Markup Language (KGML), Systems Biology Markup Language (SBML), Biological Pathway Exchange (BioPAX), and plain tabular files (CSV/TSV). All parsers conform to an abstract interface: they detect the file type, parse it, and produce normalised `MetaNode` and `MetaEdge` objects. Parsing is stateless and deterministic—the same input file always produces the same output.

Format detection is based on content analysis (root XML element) rather than file extension, making the system robust to files served without standard extensions. The parser dispatcher examines each file, selects the appropriate handler, and produces a unified stream of nodes and edges that are then merged by stable identifier in the MetaStore layer. Detailed parser specifications for each format are provided in Appendix A.

4 Storage and Indexing

The dual-layer storage architecture is the key to MetaKG's query flexibility. Rather than force all queries through a single backend (as relational databases do) or abandon structure entirely (as vector-only systems do), we maintain two complementary stores optimised for different query patterns:

SQLite Provides efficient relational queries over the graph structure. Compound–reaction–pathway relationships are naturally expressed as SQL joins. Shortest-path searches use BFS over edges. Stoichiometric detail (substrates, products, coefficients) is available as decoded JSON columns.

LanceDB + embeddings Provides semantic search over node descriptions. Users can query by natural language ("fatty-acid oxidation"), and the system returns semantically similar pathways using vector similarity. This is crucial for exploratory analysis where exact identifiers are unknown.

The two layers are not redundant; they solve fundamentally different query problems. A user might start with semantic discovery ("which pathways involve energy metabolism?"), then drill into structural detail ("what is the precise stoichiometry of ATP synthesis?"), all without leaving the interface.

4.1 SQLite Layer

Parsed nodes and edges are written to a SQLite database through the `MetaStore` class. The schema uses three tables:

`meta_nodes` One row per node. Columns correspond to the fields of `MetaNode`: **`id`**, **`kind`**, **`name`**, **`description`**, **`formula`**, **`charge`**, **`ec_number`**, **`stoichiometry`** (JSON), **`xrefs`** (JSON), **`source_format`**, **`source_file`**. Indexed on **`kind`** and **`name`**.

`meta_edges` One row per directed edge: **`src`**, **`rel`**, **`dst`**, **`evidence`** (JSON). Indexed on **`src`**, **`dst`**, and **`rel`**.

`xref_index` A materialised inverse mapping from each external identifier to the corresponding internal node ID, built after all nodes are written. This allows look-up by KEGG compound ID, ChEBI accession, UniProt accession, or any other cross-reference stored in the **`xrefs`** JSON blob.

SQLite is opened with write-ahead logging (`WAL` journal mode) and `NORMAL` synchronisation; these pragmas give throughput close to an in-memory database while retaining crash safety for workloads that write once and read many times.

4.2 Semantic Index

Structural queries are insufficient for exploratory use cases where the user does not know an exact identifier. MetaKG therefore maintains a vector index over node descriptions using LanceDB [LanceDB Contributors, 2023] as the approximate nearest-neighbour engine and the `all-MiniLM-L6-v2` sentence-transformer model [Reimers and Gurevych, 2019] as the default encoder. This model produces 384-dimensional embeddings and requires approximately 100 MB of disk space on first download.

Each node to be indexed is serialised to an embedding text that concatenates its name, molecular formula (compounds), EC number (enzymes), cross-reference values, and free-text description. Reactions are excluded from the vector index because reaction identity is better captured by their stoichiometric connectivity, which is available through the SQLite layer. Compounds, enzymes, and pathways are indexed.

The `MetaIndex.search(query, k)` method embeds the query string with the same model and returns the `k` approximate nearest neighbours, together with their cosine distances. Results are then joined against SQLite to return full node metadata.

Users may substitute any encoder that implements the `Embedder` abstract class, which requires only a single method:

```python
class Embedder(Protocol):
    def encode(self,
               texts: list[str]
               ) -> list[list[float]]: ...
```

5 Query API

MetaKG exposes four primary query modalities through a unified API. These operations work on the same graph but use different access patterns optimised for different analysis tasks. Together, they enable both precision (exact structural queries) and exploration (semantic discovery).

5.1 Node Retrieval and Neighbourhood Traversal

`MetaStore.node(id)` fetches a single node by internal ID. The returned dictionary mirrors the `meta_nodes` schema with the `xrefs` and `stoichiometry` fields pre-decoded from JSON.

`MetaStore.neighbours(id, rels=...)` returns all nodes reachable from the given node by following the specified relation types. The default relation tuple is `(SUBSTRATE_OF, PRODUCT_OF, CATALYZES, CONTAINS)`. A single SQL query over the `meta_edges` table resolves both outgoing and incoming edges and joins the results against `meta_nodes`.

5.2 Reaction Detail

`MetaStore.reaction_detail(id)` assembles a structured view of a single reaction, returning a dictionary with keys `substrates`, `products`, `enzymes`, `inhibitors`, `activators`, and `pathways`. Each value is a list of node dictionaries. This is the primary access point for stoichiometric analysis.

5.3 Shortest-Path Search

`MetaStore.find_shortest_path(a, b, max_hops)` implements an iterative breadth-first search over the SQLite graph, alternating between `SUBSTRATE_OF`/`PRODUCT_OF` edges to traverse the bipartite compound–reaction graph. The search terminates when the target node is reached or when the hop limit is exceeded. Both internal IDs and external identifiers (resolved through `xref_index`) are accepted as arguments.

The algorithm operates entirely in Python with SQL queries per BFS frontier, which is practical for graphs of the size typical of a single organism's curated metabolic network (tens of thousands of nodes). For graph corpora numbering in the hundreds of thousands of reactions, a more specialised graph engine would be preferable.

5.4 Semantic Search

`MetaKG.query_pathway(name, k)` performs a vector search over the LanceDB index and filters the results to the `pathway` node kind. The raw `MetaIndex.search(query, k)` method returns hits from all indexed kinds. Both methods accept free-text queries written in natural language; the sentence-transformer model handles biological terminology competently because the underlying training corpus includes scientific text.

`MetaKG.resolve_id(s)` provides a unified look-up that accepts any of: an internal ID (`cpd:kegg:C00022`), a shorthand external reference (`kegg:C00022`), or a display name (`Pyruvate`). It queries `xref_index` first for exact matches, then falls back to a case-insensitive name search in `meta_nodes`. This allows the same user-facing functions to accept identifiers from any source database.

5.5 Metabolic Simulations

Building on the structural query layer, MetaKG provides three simulation modalities:

1. **Flux Balance Analysis (FBA)** — `MetaKG.simulate_fba(pathway_id, maximize=True)` performs steady-state optimization using the stoichiometry stored in the graph. The result includes a status flag, objective value, and per-reaction flux distribution.
2. **Kinetic ODE Integration** — `MetaKG.simulate_ode(pathway_id, t_end, t_points, ...)` performs time-course simulation using Michaelis-Menten rate laws and seeded kinetic parameters (Km, Vmax, kcat). Uses an implicit stiff solver (BDF) optimized for metabolic systems; returns time-course concentration trajectories for all compounds.
3. **What-If Perturbation Analysis** — `MetaKG.simulate_whatif(pathway_id, scenario_json, mode)` compares baseline vs. perturbed pathway by knockout, inhibition, or substrate concentration override. Available in both FBA and ODE modes.

Kinetic parameters are populated on first use by seeding from literature sources (BRENDA, SABIO-RK, published metabolic models) via `MetaKG.seed_kinetics()`.

6 Visualisation

6.1 2D Web Explorer

The `metakg-viz` command launches a Streamlit [Streamlit Inc., 2019] web application that presents three views:

1. **Graph Browser** — an interactive network rendered with pyvis [West, 2021] and displayed in the browser. Nodes are colour-coded by kind (pathway: blue, reaction: red, compound: green, enzyme: orange) and edges are colour-coded by relation type. A sidebar allows filtering by node kind and limiting the number of rendered nodes to keep the visualisation tractable for large graphs.
2. **Semantic Search** — a free-text query box that calls `MetaIndex.search` and displays ranked results with similarity scores.

3. **Node Details** — clicking any node populates a details panel showing all node metadata and its immediate neighbourhood.

6.2 3D Visualiser

The `metakg-viz3d` command launches a PyVista [Sullivan and Kaszynski, 2019] interactive 3D viewer. Two layout strategies are implemented:

Allium layout. Each pathway node is placed at a position on a flat Fibonacci annulus in the XY-plane [Vogel, 1979]. Reaction and compound nodes belonging to a pathway are placed on a Fibonacci sphere centred on the pathway's position, producing a visual metaphor of an inflorescence. Nodes that belong to multiple pathways are placed at the centroid of their pathway positions, so cross-pathway metabolites appear in intermediate positions.

LayerCake layout. Nodes are stratified by kind along the Z-axis: pathway nodes occupy the lowest layer, reaction nodes the middle layer, and compound and enzyme nodes the upper layer. Within each layer, nodes are distributed using a golden-angle spiral to minimise overlap. This layout is better suited for inspecting the bipartite structure of the compound–reaction graph.

Both layouts export to HTML (for inclusion in web reports) and PNG (for publication figures).

7 Model Context Protocol Server

The Model Context Protocol (MCP) [Anthropic, 2024b] is a lightweight JSON-RPC standard that allows large-language-model assistants to call typed tool functions. MetaKG implements an MCP server that exposes the knowledge graph through four tools:

`query_pathway(name, k)` Semantic pathway search. Returns up to `k` pathway nodes whose descriptions are closest to the query in embedding space, together with their member-reaction counts.

`get_compound(id)` Returns a compound node with its connected reactions, accepting any supported identifier format.

`get_reaction(id)` Returns full stoichiometric detail for one reaction.

`find_path(a, b, max_hops)` Returns the shortest metabolic path between two compounds.

The server is started with:

```
metakg-mcp --db .metakg/meta.sqlite \
           --transport stdio
```

and communicates over standard input/output (the `stdio` transport) or as an HTTP server-sent events stream (the `sse` transport). The `stdio` transport is the standard configuration for use with Claude [Anthropic, 2024a] and compatible MCP clients.

8 Worked Example: Complete Human Metabolome

We demonstrate MetaKG on the complete *Homo sapiens* metabolome: all 369 KEGG pathways (metabolic, signaling, and regulatory). This corpus includes central carbon metabolism (glycolysis/gluconeogenesis, TCA cycle, pentose phosphate pathway, fatty acid degradation, oxidative phosphorylation), amino acid and nucleotide metabolism, secondary metabolism, carbohydrate metabolism, lipid metabolism, and signaling networks.

8.1 Building the Knowledge Graph

Listing 3: Building the complete human metabolome knowledge graph.

```
$ metakg-build --data ./data/hsa_pathways --wipe
Building MetaKG from ./data/hsa_pathways...
data_root   : ./data/hsa_pathways
db_path     : .metakg/meta.sqlite
nodes       : 22290  {'compound': 5115, 'enzyme': 14667,
              'pathway': 369, 'reaction': 2139}
edges       : 11298  {'CATALYZES': 2406, 'CONTAINS': 3809,
              'PRODUCT_OF': 2532, 'SUBSTRATE_OF': 2551}
indexed     : 20151 vectors  dim=384
```

The -wipe flag clears any prior database before parsing; omitting it allows incremental additions of new pathway files to an existing graph.

8.2 Structural Queries via the Python API

```python
from metakg import MetaKG

kg = MetaKG()

# Retrieve pyruvate and its connected reactions
cpd = kg.get_compound("cpd:kegg:C00022")
print(cpd["name"])
for rxn in cpd["reactions"]:
    print(f"  {rxn['name']:30s} {rxn['role']}")

# Full reaction detail
rxn = kg.get_reaction("rxn:kegg:R00200")
print(f"Substrates: {[s['name'] for s in rxn['substrates']]}")
print(f"Products:   {[p['name'] for p in rxn['products']]}")
```

Listing 4: Compound retrieval and neighbourhood traversal.

Output:

```
Pyruvate
  R00703                         SUBSTRATE_OF
  R00014                         SUBSTRATE_OF
  R00431                         SUBSTRATE_OF
  R00209                         SUBSTRATE_OF
  R00200                         PRODUCT_OF
  ... and 5 more
Substrates: ['Phosphoenolpyruvate', 'ADP']
Products:   ['Pyruvate', 'ATP']
```

8.3 Shortest-Path Search

```python
from metakg import MetaKG

kg = MetaKG()

result = kg.find_path(
    "cpd:kegg:C00031",   # D-Glucose
    "cpd:kegg:C00022",   # Pyruvate
    max_hops=12,
)
print(f"Path length: {result['hops']} steps")
for node in result["path"]:
    print(f"  {node['kind']:10s} {node['name']}")
```

Listing 5: Finding the shortest metabolic route between two compounds.

Output:

```
Path length: 9 steps
  compound   C00031
  reaction   R00299
  compound   C00092
  reaction   R00771
  compound   C00085
  reaction   R00756
  compound   C00354
  reaction   R01068
  compound   C00118
  reaction   R01061
  compound   C00236
  reaction   R01662
  compound   C01159
  reaction   R09532
  compound   C00631
  reaction   R00658
  compound   C00074
  reaction   R00200
  compound   C00022
```

The query resolves in milliseconds on the local SQLite index. The algorithm scales efficiently to the complete human metabolome (22,290 nodes, 11,298 edges) using bidirectional BFS and early termination at the target. Typical shortest-path queries complete in 10–50 ms.

8.4 Semantic Search

```python
from metakg import MetaKG

kg = MetaKG()

result = kg.query_pathway("fatty acid beta-oxidation", k=5)
for hit in result.hits:
    print(f"{hit['name']:40s}  dist={hit['_distance']:.3f}")
```

Listing 6: Semantic pathway retrieval.

Output:

```
Fatty acid degradation                    dist=1.174
alpha-Linolenic acid metabolism           dist=1.183
Fatty acid metabolism                     dist=1.200
Biosynthesis of unsaturated fatty acids   dist=1.245
Linoleic acid metabolism                  dist=1.252
```

The semantic search correctly identifies pathways related to the query, despite differences in nomenclature and terminology between the query string and the KEGG pathway names. The vector similarity is computed using the sentence-transformer model, which handles synonymy and domain terminology competently.

8.5 Pathway Analysis Report

The `metakg-analyze` command runs a seven-phase analysis and produces a structured Markdown report:

```
$ metakg-analyze --output analysis.md --top 10
```

The report covers: (1) aggregate graph statistics; (2) hub metabolites ranked by degree; (3) reactions ranked by stoichiometric complexity; (4) cross-pathway hub detection; (5) pairwise pathway coupling by shared metabolites; (6) topological features (dead-end compounds, isolated nodes); and (7) enzymes ranked by reaction count. On the complete human metabolome (369 pathways), ATP, NAD$^+$, coenzyme A, and pyruvate appear as the top hub metabolites by degree, consistent with their known roles as central energy and carbon carriers. The analysis reveals cross-pathway metabolite connectivity and identifies reactions with highest stoichiometric complexity.

8.6 Mixed-Format Ingestion

To illustrate multi-format ingestion, one may combine KGML files with a BioPAX export from Reactome and a custom CSV table in a single data directory:

```
$ ls pathways/
hsa00010.xml    # KGML (KEGG)
R-HSA-70171.owl # BioPAX (Reactome)
custom_rxns.csv # CSV (in-house data)

$ metakg-build --data ./pathways --wipe
```

The parser dispatcher examines the root XML element of each `.xml` or `.owl` file and selects the appropriate parser; `.csv` and `.tsv` files are handled by the tabular parser. Nodes that share a KEGG or ChEBI cross-reference across files are automatically merged in SQLite through the xref index.

9 Implementation Notes

Dependencies. The core package requires Python 3.10–3.12, `lancedb` ≥ `0.29`, `numpy` ≥ `1.24`, and `sentence-transformers` ≥ `2.7`. No network connection is required at run time once the embedding model has been downloaded. BioPAX support requires the optional `rdflib` package; the 2D and 3D visualisers require optional extras installed via `poetry install -extras viz` or `-extras viz3d`.

Installation.

```
git clone https://github.com/Flux-Frontiers/meta_kg
cd meta_kg
poetry install --extras all
```

Thread safety. The SQLite connection uses WAL mode with a single Python object per process. The current implementation is not thread-safe; callers should create one `MetaKG` instance per process or protect the shared instance with a lock.

Incremental rebuild. The stable ID scheme makes incremental builds tractable. Running `metakg-build` without `-wipe` issues `INSERT OR REPLACE` statements, which update existing nodes and append new ones without duplicating entries.

Codebase self-analysis. MetaKG is itself analysed with its sister tool CodeKG [Flux Frontiers Contributors, 2025], which constructs a structural and semantic knowledge graph of the Python source code. This enables navigating the MetaKG implementation via natural-language queries and validates that the two tools share compatible architectural patterns. On the MetaKG codebase the CodeKG static analysis produces 3,136 nodes and 2,920 edges spanning 27 modules, embedded into a 384-dimensional vector index of 290 vectors.

10 Discussion

10.1 Architectural Design Choices and Novel Aspects

MetaKG embodies several deliberate design trade-offs that distinguish it from existing systems. First, the dual-layer architecture (SQLite for structure, LanceDB for semantics) is novel in the metabolic pathway domain. Graph databases like Neo4j support complex queries but introduce operational overhead (server management, scaling, deployment) and do not address the multi-format parsing problem. Specialised vector databases like Weaviate or Pinecone excel at semantic search but are not natural for structural graph traversal and require network access. By combining lightweight SQLite with local vector search, MetaKG provides both capabilities in a self-contained package suitable for exploratory research and reproducible analysis workflows.

Second, the deterministic identifier scheme with synthetic hashing enables reproducible cross-format merging without a centralised reconciliation service. Unlike MetaNetX (which requires API calls to reconcile identifiers), MetaKG builds self-contained graphs. This design makes the graph a version-controlled artefact: the same input files always produce the same output graph, enabling reproducible science and offline workflows.

Third, the four-modality query interface (structural, pathfinding, semantic, stoichiometric) is intentionally broad. Analysts can start with a natural-language semantic query ("fatty-acid beta-oxidation"), then drill into structural detail (shortest paths, stoichiometric coefficients) without leaving the interface. This contrasts with systems that specialise in one query paradigm.

Fourth, the Model Context Protocol integration is forward-looking. As large-language-model assistants become standard tools in computational biology, making the knowledge graph a first-class data source for Claude, ChatGPT, and future assistants is a natural evolution. The MCP interface is not merely an API; it represents a design commitment to AI-accessible knowledge graphs.

10.2 Scope and Design Trade-offs

Snapshot-based operation. MetaKG is a local analysis tool, not a live database mirror. The knowledge graph is a snapshot at build time; users must re-run `metakg-build` after updating source files. This design choice keeps the system self-contained and avoids dependencies on external services during analysis. For research workflows where reproducibility is paramount, this is an advantage. For applications requiring real-time data updates, a live database backend would be preferable.

Scale. SQLite and in-process BFS are appropriate for graphs of up to roughly 100,000 nodes, covering full reconstructed metabolic networks for a single organism. For pan-genome or multi-species analyses—where node counts reach into the millions—a dedicated graph database engine (e.g., Neo4j [Robinson et al., 2015] or Kùzu [Feng et al., 2023]) and a distributed vector index would be preferable. The storage layer is designed to be replaceable: `MetaStore` and `MetaIndex` are concrete classes behind well-defined interfaces, and alternative backends could be substituted without changing the parser or query layers.

Identifier reconciliation. The current cross-reference merge is based on exact match of external identifiers. Two compounds that share a biological identity but differ in stereochemistry or protonation state will not be automatically merged. More complete reconciliation would require integration with a name normalisation service such as MetaNetX [Moretti et al., 2021] or UMLS [Bodenreider, 2004].

Stoichiometric models and simulations. MetaKG provides three core simulation modalities: (1) Flux Balance Analysis (FBA) for steady-state flux optimization; (2) kinetic ODE integration with Michaelis-Menten rate laws using an implicit stiff solver (BDF) optimized for metabolic pathways; and (3) what-if perturbation analysis for enzyme knockouts, inhibition, and substrate overrides. Kinetic parameters are seeded from literature (BRENDA, SABIO-RK, published models). For advanced constraint-based modelling workflows, the SBML parser preserves all information needed to reconstruct a COBRApy model.

Performance on complete human metabolome. The complete *Homo sapiens* metabolome (369 KEGG pathways, 22,290 nodes, 11,298 edges, 20,151 vector embeddings) exhibits the following performance characteristics:

Table 2: Performance on the complete human metabolome (22K nodes, 11K edges).

Operation	Time	Details
Build graph (parse + index)	30–60s	All 369 pathways, LanceDB + SQLite
Semantic search (natural language)	100–500ms	Vector similarity on 20K nodes
Shortest-path (6 hops)	10–50ms	BFS on 11K edges
ODE simulation (10 units)	150–400ms	BDF solver, 24 compounds
Streamlit rerun	0.5–1.5s	Batch query + session cache

All operations complete within practical timeframes for interactive exploration. The graph is suitable for research workflows requiring reproducible analysis, offline access, and programmatic querying.

Future directions. Planned extensions include: graph-theoretic centrality measures (betweenness, closeness, eigenvector centrality) implemented with NetworkX for hub ranking; a GraphQL query endpoint; integration with the UniProt and ChEBI REST APIs for on-demand annotation; differential pathway analysis across two or more organisms; GPU-accelerated embedding for large datasets; and export to Cytoscape [Shannon et al., 2003] JSON for interoperability with the broader network biology ecosystem. Advanced kinetic parameter optimization (parameter fitting from experimental data) is also planned.

11 Conclusion

MetaKG addresses a fundamental gap in metabolic data integration: existing systems force a choice between convenient web interfaces (limited programmability, no semantic search, web-dependent), specialised graph databases (high operational complexity, parsing not solved), and reconciliation services (queryable graphs not provided). MetaKG breaks this false dichotomy by introducing a dual-layer local knowledge graph that unifies multi-format pathway data and enables four orthogonal query modalities—structural, pathfinding, semantic, and stoichiometric—all through a single API, CLI, and LLM-accessible MCP interface.

The core contributions are: (1) a stable, deterministic identifier scheme that enables reproducible cross-format merging without external services; (2) a dual-layer storage architecture (SQLite + LanceDB) that avoids the false choice between relational precision and semantic expressivity; (3) four unified query modalities accessible to analysts at all levels of programming expertise; and (4) a forward-looking MCP server that makes metabolic knowledge graphs first-class data sources for AI assistants.

The design philosophy—local-first, self-contained, snapshot-based—is intentional and represents a clean separation from live database mirrors. This makes MetaKG particularly well-suited for research workflows where reproducibility, offline analysis, and version control are paramount. For applications requiring real-time data, larger graph corpora, or distributed deployment, the modular architecture enables substitution of the storage and index backends.

We expect MetaKG to be immediately useful as: (1) a foundation for pathway analysis scripts; (2) a data preparation stage for machine-learning workflows; (3) an AI-accessible knowledge source for metabolic reasoning tasks in large-language-model applications; and (4) a template for similar knowledge graph systems in related biological domains (protein interactions, gene regulatory networks, drug-target interactions).

The software is freely available at `https://github.com/Flux-Frontiers/meta_kg` under the Elastic License 2.0.

A Format-Specific Parsers

All parsers conform to an abstract base class:

```
class PathwayParser:
    def can_handle(self, path: Path) -> bool: ...
    def parse(self, path: Path
             ) -> tuple[list[MetaNode],
                        list[MetaEdge]]: ...
```

Parsers are stateless and pure: the same input file always produces the same output. The `MetabolicGraph` layer caches the combined node and edge lists after the first parse, so repeated calls do not re-read disk.

A.1 KGML Parser

KEGG Markup Language files are the native export format of the KEGG pathway database [Kanehisa et al., 2021]. Each file is an XML document whose root element is `<pathway>`. The parser uses the Python standard-library `ElementTree` module (no third-party XML dependency) and extracts three kinds of child elements:

- `<entry>` elements with `type="compound"` become `compound` nodes.
- `<entry>` elements with `type="gene"` or `type="enzyme"` become `enzyme` nodes.
- `<reaction>` elements become `reaction` nodes with their `<substrate>` and `<product>` children encoded as stoichiometry JSON. The enclosing pathway becomes a `CONTAINS` edge to each reaction.

Format detection is based on the root element tag rather than the file extension, making the parser robust to KEGG files that are served without the `.kgml` extension.

A.2 SBML Parser

The Systems Biology Markup Language [Keating et al., 2020] is the standard serialisation format for constraint-based metabolic models generated by tools such as COBRApy [Ebrahim et al., 2013]. SBML Level 2 and 3 files share a common XML namespace ending in `sbml`; the parser detects format by matching the root element's local name.

Species elements map to `compound` nodes. Reaction elements map to `reaction` nodes. Stoichiometry is extracted from `<listOfReactants>` and `<listOfProducts>` children. Modifier species are classified by their SBO term [Courtot et al., 2011]: SBO:0000013 (catalyst) generates `CATALYZES` edges; SBO:0000020 (inhibitor) generates `INHIBITS` edges; other modifiers generate `ACTIVATES` edges.

A.3 BioPAX Parser

Biological Pathway Exchange Level 3 [Demir et al., 2010] is an OWL ontology serialised as RDF/XML that is used by Reactome [Jassal et al., 2020], WikiPathways [Martens et al., 2021], and the NCI Pathway Interaction Database. Parsing requires the optional `rdflib` dependency, which is installed via the `biopax` extra. The parser performs SPARQL-style pattern matching over the RDF graph to extract:

- `SmallMolecule` instances → `compound` nodes.
- `Protein` instances → `enzyme` nodes.
- `BiochemicalReaction` instances → `reaction` nodes, with `left`/`right` properties becoming substrate and product edges, and `controller` properties becoming `CATALYZES` edges.
- `Pathway` instances → `pathway` nodes, with `memberPathwayComponent` links becoming `CONTAINS` edges.

A.4 CSV/TSV Parser

For custom or unpublished data, MetaKG accepts flat tables with a configurable column schema. The default column layout is:

`reaction_id`, `reaction_name`, `substrate`, `product`, `enzyme`, `stoich_substrate`, `stoich_product`, `pathway`, `ec_number`, `substrate_formula`, `enzyme_uniprot`

Multiple rows with the same `reaction_id` are merged into a single reaction node, which is the standard way to encode multi-substrate or multi-product reactions in tabular form. A `CSVParserConfig` dataclass allows remapping all column names, making the parser suitable for lab-produced spreadsheets and bulk downloads from custom databases.

References

Anthropic. Claude: AI assistant. https://claude.ai, 2024a. Accessed February 2026.

Anthropic. Model Context Protocol (MCP): an open protocol that enables seamless integration between LLM applications and external data sources and tools. https://modelcontextprotocol.io, 2024b. Accessed February 2026.

Olivier Bodenreider. The Unified Medical Language System (UMLS): integrating biomedical terminology. *Nucleic Acids Research*, 32(suppl_1):D267–D270, 2004. doi: 10.1093/nar/gkh061.

Mélanie Courtot, Nick Juty, Christian Knüpfer, et al. Controlled vocabularies and semantics in systems biology. *Molecular Systems Biology*, 7:543, 2011. doi: 10.1038/msb.2011.77.

Emek Demir, Michael P. Cary, Suzanne Paley, et al. The BioPAX community standard for pathway data sharing. *Nature Biotechnology*, 28(9):935–942, 2010. doi: 10.1038/nbt.1666.

Ali Ebrahim, Joshua A. Lerman, Bernhard Ø. Palsson, and Daniel R. Hyduke. COBRApy: COnstraints-Based Reconstruction and Analysis for Python. *BMC Systems Biology*, 7:74, 2013. doi: 10.1186/1752-0509-7-74.

Xiyang Feng, Guodong Jin, Ziyi Chen, Chang Liu, and Semih Salihoğlu. KÙZU Graph Database Management System. In *Proceedings of the VLDB Endowment*, volume 16, pages 3630–3637, 2023. doi: 10.14778/3611540.3611556.

Flux Frontiers Contributors. CodeKG: A Structural and Semantic Knowledge Graph for Python Codebases. https://github.com/Flux-Frontiers/code_kg, 2025. Accessed February 2026.

Bijay Jassal, Lisa Matthews, Guilherme Viteri, et al. The reactome pathway knowledgebase. *Nucleic Acids Research*, 48(D1):D498–D503, 2020. doi: 10.1093/nar/gkz1031.

Minoru Kanehisa, Miho Furumichi, Yoko Sato, Mari Ishiguro-Watanabe, and Masumi Tanabe. KEGG: integrating viruses and cellular organisms. *Nucleic Acids Research*, 49(D1):D545–D551, 2021. doi: 10.1093/nar/gkaa970.

Sarah M. Keating, Dagmar Waltemath, Matthias König, et al. SBML Level 3: an extensible format for the exchange and reuse of biological models. *Molecular Systems Biology*, 16(8):e9110, 2020. doi: 10.15252/msb.20199110.

LanceDB Contributors. LanceDB: Developer-friendly, serverless vector database. https://lancedb.github.io/lancedb/, 2023. Accessed February 2026.

Marvin Martens, Ammar Ammar, Anders Riutta, et al. WikiPathways: connecting communities. *Nucleic Acids Research*, 49(D1):D613–D621, 2021. doi: 10.1093/nar/gkaa1024.

Sébastien Moretti, Kristian Paskov, Richard H. R. Hahnloser, et al. MetaNetX/MNXref: unified namespace for metabolites and biochemical reactions in the context of metabolic models. *Nucleic Acids Research*, 49 (D1):D570–D574, 2021. doi: 10.1093/nar/gkaa992.

Nils Reimers and Iryna Gurevych. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*, pages 3982–3992. Association for Computational Linguistics, 2019. doi: 10.18653/v1/D19-1410.

Ian Robinson, Jim Webber, and Emil Eifrem. *Graph Databases: New Opportunities for Connected Data*. O'Reilly Media, 2nd edition, 2015. ISBN 978-1-491-93089-2.

Paul Shannon, Andrew Markiel, Owen Ozier, et al. Cytoscape: a software environment for integrated models of biomolecular interaction networks. *Genome Research*, 13(11):2498–2504, 2003. doi: 10.1101/gr.1239303.

Streamlit Inc. Streamlit: The fastest way to build data apps. https://streamlit.io, 2019. Accessed February 2026.

C. Bane Sullivan and Alexander A. Kaszynski. PyVista: 3D plotting and mesh analysis through a streamlined interface for the Visualization Toolkit (VTK). *Journal of Open Source Software*, 4(37):1450, 2019. doi: 10.21105/joss.01450.

Helmut Vogel. A better way to construct the sunflower head. *Mathematical Biosciences*, 44(3-4):179–189, 1979. doi: 10.1016/0025-5564(79)90080-4.

Jason West. Pyvis: visualize and construct interactive network graphs in Python. https://pyvis.readthedocs.io, 2021. Accessed February 2026.