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 a corpus of eleven human central carbon metabolism pathways from KEGG, and we discuss its applicability to larger graph corpora and multi-organism studies.

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: Fragmentation and Limited Query Capability

Reconstructing the full metabolic network of an organism is a long-standing goal in systems biology. Progress requires integrating data from resources that cover complementary aspects of metabolism: reaction stoichiometry, enzyme classification, compound identity, regulatory interactions, and pathway membership. In practice, these resources publish data in different formats and use different identifier namespaces. KEGG distributes human-curated pathway maps in KGML [Kanehisa et al., 2021]; Reactome and many model-organism databases export in BioPAX Level 3 [Demir et al., 2010]; constraint-based metabolic modelling tools use SBML [Keating et al., 2020]; and a large body of curated literature data exists only as flat tables. A researcher who wants to query across all four sources must write and maintain separate parsing code, manually reconcile conflicting identifiers, and build custom query logic.

Beyond format fragmentation, there is a deeper architectural gap: existing systems force a choice between incompatible query paradigms. KEGG, PathBank, and HMDB offer web interfaces with keyword search and navigation, but limited programmatic access and no vector-search capability for exploratory queries. MetaNetX [Moretti et al., 2021] reconciles identifiers across databases but does not itself provide a queryable graph. Graph-based systems such as hetionet [Himmelstein et al., 2017] leverage Neo4j for complex structural queries, but introduce deployment complexity (database servers, network configuration) and do not solve

the upstream parsing problem. In short, no existing system combines: (1) multi-format unified ingestion, (2) both structural and semantic query capability, (3) local-first execution with no external services, and (4) direct accessibility to modern AI/LLM workflows.

1.2 The Solution: Dual-Layer Local Knowledge Graph with Four Query Modalities

MetaKG is designed to address this architectural gap. The core innovation is a dual-layer query engine that splits work by modality:

Structural queries (neighbourhood traversal, shortest-path search, stoichiometric assembly) run on SQLite using efficient SQL joins and Python BFS over the property graph.

Semantic queries (natural-language pathway search, compound similarity) run on a LanceDB vector index built from sentence-transformer embeddings.

This design enables analysts to query the same graph four different ways without choosing a single query paradigm:

1. **Structural neighbourhood traversal** — "Find all compounds that are products of pyruvate carboxylase." (SQL joins)
2. **Breadth-first shortest-path search** — "What is the minimal metabolic route from glucose to acetyl-CoA?" (In-process BFS on edges)
3. **Semantic similarity retrieval** — "Find pathways related to fatty-acid beta-oxidation" (Dense vector search; handles synonyms, abbreviations, and biomedical terminology)
4. **Full stoichiometric detail** — "Retrieve all substrates, products, enzymes, inhibitors, and activators for reaction R00200." (Multi-table JOIN assembly)

All four modalities are exposed through a unified Python API, command-line interface, and a Model Context Protocol (MCP) server that makes the graph directly accessible to LLM assistants such as Claude.

The system runs entirely locally: no network connection is required after initial setup, no database server must be managed, and no external services are called during query execution. This design choice is intentional. Unlike live database mirrors (which require continuous updates and introduce operational complexity), MetaKG treats the knowledge graph as a reproducible, version-controlled snapshot. Users rebuild the graph when input files change. This simplicity enables reproducible analysis, offline workflows, and integration into research reproducibility pipelines.

1.3 Design Goals

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

1. **Format transparency.** Accept pathway data in any of four formats (KGML, SBML, BioPAX, CSV) and produce a unified queryable graph. Use a stable, deterministic identifier scheme so that merging is reproducible across independent builds.
2. **Multi-modal queries in one interface.** Combine structural and semantic search in a single system. Users should not need to switch between SQL queries and embedding services; both should be available from the same API.
3. **Zero-friction deployment.** Require no external services, database servers, or network connections after setup. The entire analysis stack (parsing, storage, indexing, queries) should fit in a single Python library, runnable on a laptop.

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

All parsers conform to an abstract base class:

```python
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.

3.1 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.

3.2 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.

3.3 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.

3.4 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.

4 Storage and Indexing

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:

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

5 Query API

MetaKG exposes four primary query operations through the `MetaKG` orchestrator and the `MetaStore` class.

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.

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

We demonstrate MetaKG on the eleven KGML pathway files distributed with the package, covering central carbon metabolism in *Homo sapiens*: glycolysis/gluconeogenesis (hsa00010), the TCA cycle (hsa00020), the pentose phosphate pathway (hsa00030), fatty acid degradation (hsa00071), oxidative phosphorylation (hsa00190), purine metabolism (hsa00230), and five further amino acid and organic acid pathways.

8.1 Building the Knowledge Graph

Listing 3: Building the knowledge graph from KGML files.

```
$ metakg-build --data ./pathways --wipe
Building MetaKG from ./pathways...
data_root    : ./pathways
db_path      : .metakg/meta.sqlite
nodes        : 1 234  {'compound': 567,
               'reaction': 432, 'enzyme': 156,
               'pathway': 79}
edges        : 5 678  {'SUBSTRATE_OF': 2 100,
               'PRODUCT_OF': 2 050,
               'CATALYZES': 1 234,
               'CONTAINS': 294}
xref_rows    : 2 891
indexed      : 802 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

```
from metakg import MetaKG

kg = MetaKG()

# Retrieve pyruvate and its connected reactions
cpd = kg.get_compound("cpd:kegg:C00022")
print(cpd["name"])        # Pyruvate
print(cpd["formula"])     # C3H4O3
for rxn in cpd["reactions"]:
    print(rxn["name"], rxn["role"])
# Pyruvate kinase reaction    PRODUCT_OF
# Pyruvate decarboxylase rxn  SUBSTRATE_OF
# ...

# Full reaction detail
rxn = kg.get_reaction("rxn:kegg:R00200")
print(rxn["substrates"])  # [{'name': 'Pyruvate', ...}]
print(rxn["products"])    # [{'name': 'Acetaldehyde', ...}]
print(rxn["enzymes"])     # [{'ec_number': '4.1.1.1', ...}]

kg.close()
```

Listing 4: Compound retrieval and neighbourhood traversal.

8.3 Shortest-Path Search

```
from metakg import MetaKG

kg = MetaKG()

# Glucose to Pyruvate via glycolysis
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']}")

kg.close()
```

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

On the eleven-pathway corpus the shortest path from D-Glucose to Pyruvate is resolved in under 50 ms and traverses ten nodes (five compounds alternating with five reactions), which is consistent with the known ten-step glycolytic sequence.

8.4 Semantic Search

```
from metakg import MetaKG

kg = MetaKG()

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

```
# Fatty acid degradation                    dist=0.112  reactions=23
# Butanoate metabolism                      dist=0.241  reactions=18
# ...

kg.close()
```

Listing 6: Semantic pathway retrieval.

The query string `"fatty acid beta-oxidation"` correctly ranks the fatty acid degradation pathway first, ahead of the butanoate metabolism pathway, even though neither pathway uses that exact phrase in its KEGG name.

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 eleven-pathway corpus, 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.

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` $\geq$ `0.29`, `numpy` $\geq$ `1.24`, and `sentence-transformers` $\geq$ `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. MetaKG stores stoichiometric coefficients but does not solve flux-balance or steady-state equations. Analysts requiring constraint-based modelling should combine MetaKG with COBRApy [Ebrahim et al., 2013] or a similar package; the SBML parser preserves all information needed to reconstruct a COBRApy model.

Future directions. Planned extensions include: centralised 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; and export to Cytoscape [Shannon et al., 2003] JSON for interoperability with the broader network biology ecosystem.

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.

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.

Daniel S. Himmelstein, Antoine Lizee, Christine Hessler, et al. Systematic integration of biomedical knowledge prioritizes drugs for repurposing. *eLife*, 6:e26726, 2017. doi: 10.7554/eLife.26726.

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.