ChatGPT

# Pragmatic Graph Development: A Domain-Driven Approach

Building graph-based systems **pragmatically** means focusing on the domain's needs and data first, rather than immediately forcing everything into a generic graph database schema. This approach emphasizes designing your graph structures around real use cases and iterating on the design, before leveraging graph database technology for visualization or complex queries. In practice, **graphs can exist outside of graph databases** – even simple JSON files or relational tables can represent a "graph" of data. The key is to use the right format at each stage and only introduce complexity (like a full graph database) when it adds clear value.

## Domain-Specific Modeling vs. Generic Graph Databases

When starting a graph project, it's tempting to shove all information directly into a graph database as nodes and edges. However, a one-size-fits-all graph schema can lead to a design that doesn't truly fit the problem. A **domain-driven approach** begins by capturing data in a format that naturally reflects the domain:

- **Use Native Data Structures First:** Begin with the format that best represents your data (e.g. JSON, classes/objects, relational tables) rather than immediately defining generic graph nodes and relationships. For example, if you're modeling source code, you might start with JSON or Python classes that mirror code structure (files, classes, functions, call relationships) in a straightforward way.
- **Avoid Premature Abstraction:** Designing an abstract graph model too early can introduce unnecessary complexity. It's often more effective to first create a *specific* schema or data structure for your problem, then generalize if needed. This ensures the model captures exactly what's needed, without extraneous concepts.

This strategy aligns with best practices for graph data modeling – you should consider your specific use cases and iterate on the model rather than adopting a fixed schema upfront [1]. By deferring the use of a graph database and focusing on a **domain-specific graph structure**, you maintain clarity. The design remains closely coupled to the problem at hand, which actually makes it easier to evolve and refactor.

## Iterative Schema Design and Visualization

A hallmark of pragmatic graph development is **iterative design**. Plan to refine your data model through multiple iterations:

1. **Initial Schema Draft:** Define an initial schema or classes for your data. Don't worry if it's not perfect – include all data fields and relationships you think might be useful. At this stage, it's fine to be somewhat redundant or verbose in representation.
2. **Visual Inspection of Data:** Represent some real data in this format (for example, output it as a JSON structure). Viewing the data in a human-readable form is incredibly valuable – JSON or similar nested structures let you *see the graph* in a tree-like form. This visual feedback helps identify if the structure makes sense or if certain elements are cumbersome.

3. **Refine and Simplify:** Based on what you learn, refactor the schema. Remove or reorganize fields that are redundant or can be derived. Rename things for clarity. Possibly merge or split structures if needed. Each iteration should bring the data model closer to an optimal representation of the domain.
4. **Leverage Type Safety (if available):** If you have a system for defining classes or schemas with type safety, use it. Strongly-typed data classes can catch inconsistencies early and ensure that as you tweak the model, the data remains valid. In our example, we used custom type-safe classes to define nodes (with unique IDs, types, and fields for relationships) which helped enforce consistency.
5. **Repeat:** It often takes multiple passes (and possibly feedback from domain experts or AI assistants) to get the schema right. In one case, we went through several design reviews, each time eliminating unnecessary parts and making implicit relationships explicit only where needed. After a few iterations, the schema became much cleaner and leaner than the first draft.

Throughout this process, **JSON structures were extremely useful as a working representation**. JSON (or Python dictionaries, etc.) naturally capture hierarchical data and can even represent graph connectivity (e.g. by using IDs or references). Because JSON is human-readable, it served as a quick way to verify the correctness of the model. One could argue that a JSON document is itself a form of a graph (a tree of nested objects with references), so working in JSON means you *are* working with a graph – just in a more transparent way than a black-box graph database.

**Example – Visualizing with JSON:** In our source code graph project, we represented a module of Python code as a JSON object containing classes, functions, and their relationships (calls, imports, etc.). Seeing it in JSON made it easy to spot if, say, a function was missing a reference to its parent class or if a relationship was modeled awkwardly. This visual clarity is harder to achieve if the data is locked inside a graph database from the start.

## Integrating with Graph Databases (Round-Trip Data Flow)

After refining a domain-specific graph model, you can then harness a graph database for what it does best: **querying relationships and visualizing connections**. The pragmatic approach treats the graph database as a **secondary projection or tool** rather than the primary store (at least during design time):

- **One-Way Export:** Take your finalized schema and build a translator that converts the domain objects/JSON into nodes and edges in the graph database. For instance, you might have a function that reads your JSON and issues commands to create nodes for each entity and edges for each relationship in a graph DB like Neo4j or a custom graph engine (in our case, "MGraphDB"). This lets you run graph queries (e.g., find all functions that ultimately call a certain API, or visualize the call graph of a module) using the rich query languages and tools available.
- **Leveraging Graph DB Capabilities:** With the data loaded, you gain the benefits of the graph database: you can traverse relationships easily, run complex path queries, and generate visualizations (e.g., using graph visualization libraries or export to tools like Mermaid or D3.js for diagrams). The graph database acts as a **powerful lens** on your data, answering questions that are hard to do in the raw JSON (e.g. multi-hop traversals) and creating pretty diagrams for reports or analysis.

Crucially, this approach means **the graph database is not the single source of truth for the data** – your original domain-specific representation remains the authoritative source. This idea is echoed by others: you can keep your primary database (or data structure) as the clean source of truth, and **layer a graph representation on top for relationship-centric analysis** [2]. By doing so, you don't have to "rip

out" your existing data format; instead, you project it into a graph form when needed, and otherwise continue to use the original format for other operations (validation, editing, etc.).

- **Round-Trip (Synchronization):** Ideally, set up a round-trip so that if the graph database version is modified (e.g., you add a new node or edge via a visual interface), those changes can be pulled back into your JSON/domain model. This may be more complex, and in many cases you might use the graph DB in a read-only analytical way. But even a one-way sync (from source model to graph DB) is useful. If a round-trip is implemented, ensure that every piece of information in the source model is either stored in the graph or can be derived from it, so nothing is lost. In our project, we paid attention to **node identifiers and metadata** – every node in the JSON had a unique ID that was used as the node ID in the graph database. This way, any results or modifications from the graph can be mapped back to the original structure.
- **No Data Loss:** A good test of your integration is to be able to export to the graph DB and then re-import back to the original format without losing data. For example, we experimented with an HTML graph: converting an HTML page into a custom JSON graph structure (capturing elements, attributes, parent-child relationships), loading that into a graph database for querying, and then reconstructing the original HTML from the JSON structure. By ensuring all attributes and text were preserved in the JSON, the round-trip HTML -> JSON -> graph -> JSON -> HTML did not lose any content or structure. This kind of lossless conversion validates that your graph representation is truly an alternate view of the same data, not a destructive transformation.

It's worth noting that using a graph database is optional until you need it. Many problems can be solved with the domain model alone. But when you do need the firepower of graph queries or interactive network diagrams, having a **translation layer** to a graph DB gives you the best of both worlds.

## Case Study: Source Code Knowledge Graph

To illustrate the approach, consider a real-world scenario of building a knowledge graph for source code analysis. In this project, there were two interconnected graph structures:

- **Semantic Graph (Ontology/Taxonomy):** A graph of concepts describing code, such as categories of functions, types of relationships (calls, inherits, imports), and rules about how code elements relate (an ontology of programming concepts). This is a meta-graph that defines types and classifications.
- **Code Instance Graph (Call Flow Graph):** A graph representing the actual codebase – modules, classes, functions, and the relationships between them (function A calls function B, class X inherits class Y, module M imports module N, etc.).

A naïve approach might store everything in one big graph database from the beginning: each ontology concept, each code element, all as nodes with edges linking them. Instead, the pragmatic method was applied:

1. **Separate Domain Models:** We created a **schema for the ontology** and a **schema for the code graph**, using Python classes and JSON as the representation. For example, a `Function` class had fields like `name`, `belongs_to_class`, `calls = [list of Function IDs]`, etc., and an `OntologyNode` class had fields like `concept_name`, `subconcepts`, or rules attached. These classes were strongly typed and could serialize to JSON, so we could easily inspect the ontology hierarchy or the call flows in a JSON file.
2. **Populate and Iterate:** We populated these structures with real data (actual Python code analysis). On reviewing the JSON output, we found areas to improve. For instance, we noticed we were storing some relationships twice (once in the caller function and once in the callee, as an

example) – a sign of redundancy. We refactored the schema to store each relationship in one place and derive the inverse when needed. In another iteration, we realized certain global data (like a list of all functions) was better represented as a dictionary for quick access by ID, rather than a list that we'd search through. Each change was driven by looking at how the JSON data was being used or could be queried.

3. **Ensure Graph-Compatibility:** Even though we weren't using the graph database yet, we kept the eventual graph export in mind. Each entity was given a stable identifier (e.g., a fully-qualified name or a GUID) that would become the node ID in the graph DB. Relationships in the JSON (like function calls) were stored by referencing those IDs. This meant when it came time to export, it was straightforward to translate JSON objects to graph nodes and their references to graph edges.
4. **Export to Graph Database:** Once the models felt solid, we wrote a connector to export both graphs into MGraphDB (our graph database system). Ontology concepts became nodes (with an "is-a" or "category" edge between them to form the taxonomy tree). Code elements became nodes of type "Function", "Class", etc., with edges like "CALLS", "DEPENDS_ON", "DEFINED_IN" linking them. We could then run queries such as "find all functions ultimately related to a certain ontology concept" by traversing from code nodes up through ontology nodes. We could also generate visualizations: for example, a Mermaid.js diagram of a module's call graph, or an interactive D3.js view of the ontology, all powered by the graph database.
5. **Maintain the Source of Truth:** Despite loading data into the graph DB, the **source of truth remained the JSON and class structures**. If the code changed or the ontology was updated, we would update the JSON representation via our classes and then regenerate the graph view. This ensured consistency and leveraged the strengths of each format (JSON for easy editing and schema evolution, graph DB for complex querying and visualization).

The outcome was very successful. By not entangling the initial design with graph database specifics, we were free to evolve the model quickly. Yet, we didn't miss out on graph database benefits – we engaged them at the right time. Additionally, because of the careful mapping (like consistent IDs), we could always trace a node in the graph database back to the original JSON record or code element it represented.

## Benefits of the Pragmatic Approach

Adopting this pragmatic, domain-first approach to graph development yields several benefits:

- **Better Alignment with Requirements:** The data model is tailored to actual needs and refined with real examples, so it captures the domain accurately. You avoid shoehorning the problem into a generic graph schema that doesn't quite fit.
- **Iterative Improvement:** Because you can see and test the data in a simple format, you can iteratively improve the schema. This leads to a cleaner, more optimized design (often with redundant data eliminated and clear relationships defined) before committing it to a database schema.
- **Simplicity and Clarity:** Team members can read the JSON or class-based data structures and understand the graph model easily. This transparency can make collaboration easier, as opposed to dealing with abstract graph queries from day one.
- **Use of Best Tool for the Job:** You leverage tools for what they do best – e.g., JSON or relational structures for transactional integrity or ease of editing, and graph databases for connected queries and visualization. This combination can be more powerful than forcing everything into one tool. In fact, keeping a traditional database as the system of record and layering a graph for specialized querying is a known best practice [2].

- **Flexibility to Change Technology:** Since the core data isn't locked into a particular graph database, you have flexibility. If a new graph technology comes along, you can write a new exporter without redesigning your whole system. The domain model lives independently of any specific graph DB.
- **Round-Trip Data Consistency:** If implemented, a round-trip integration ensures that analyses or modifications done on the graph side can be brought back to the source format. Even if you don't need to modify data via the graph, knowing that the graph view could be reconstructed faithfully from the source gives confidence in the completeness of the graph representation.

In summary, **pragmatic graph development** is about being *practical* and *purposeful* with how you design and implement graph-based systems. Start with the problem and the data — model those in the simplest, most direct way. Embrace an **iterative, domain-driven modeling process** [1], and only then project the result into a graph database to leverage powerful graph capabilities. By doing so, you ensure that your graph isn't just "cool technology" but a true asset that makes sense to everyone involved and delivers real value with minimal complexity.

---


[1] Graph Data Modeling: All You Need To Know
https://www.puppygraph.com/blog/graph-data-modeling

[2] From Rows to Relationships: turn your SQL into a graph. | by Fabio Yáñez Romero | Data Science Collective | Medium
https://medium.com/data-science-collective/from-rows-to-relationships-turn-your-sql-into-a-graph-16436e9832e9