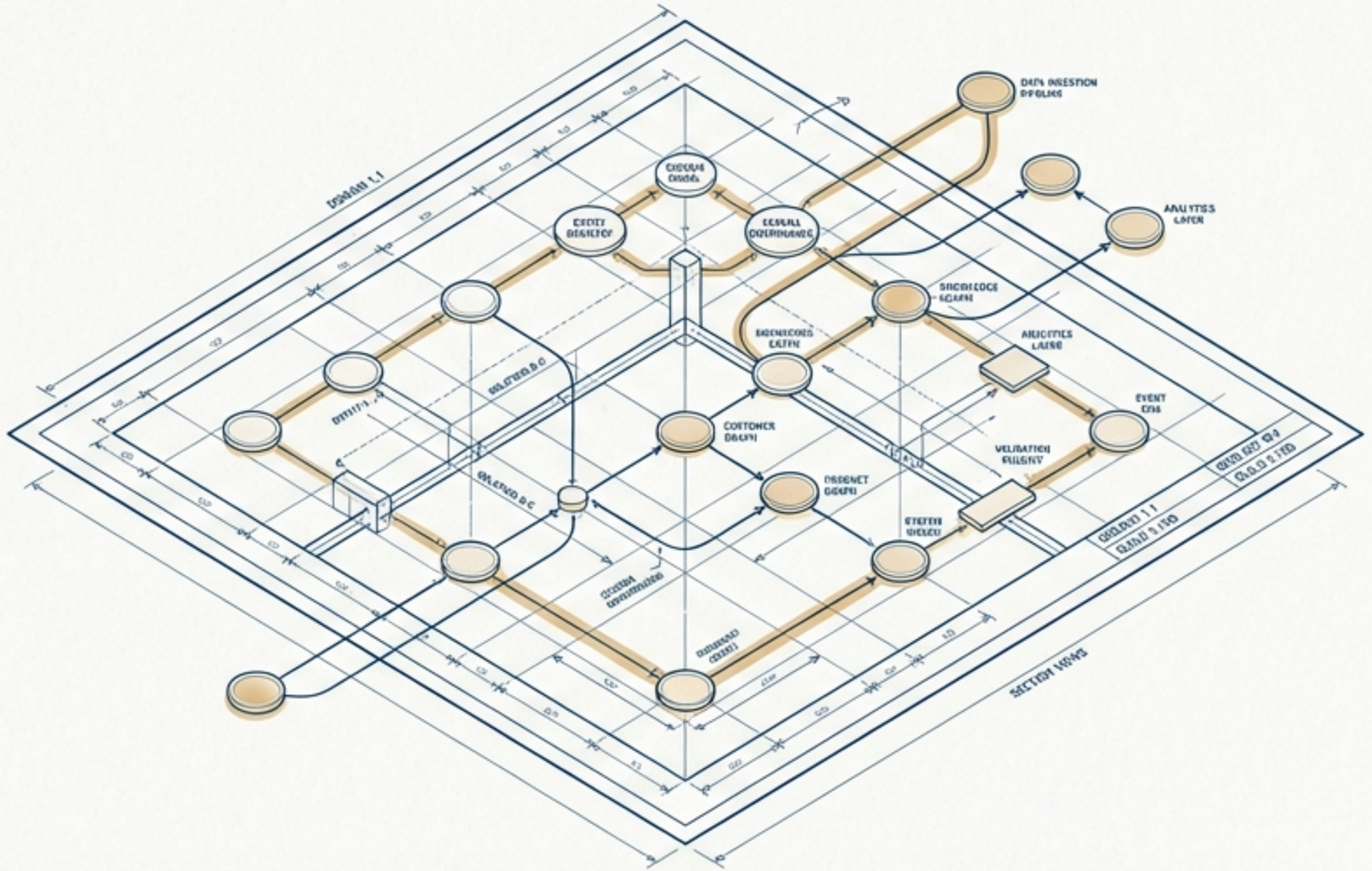
The Pragmatic Path to Graph Development

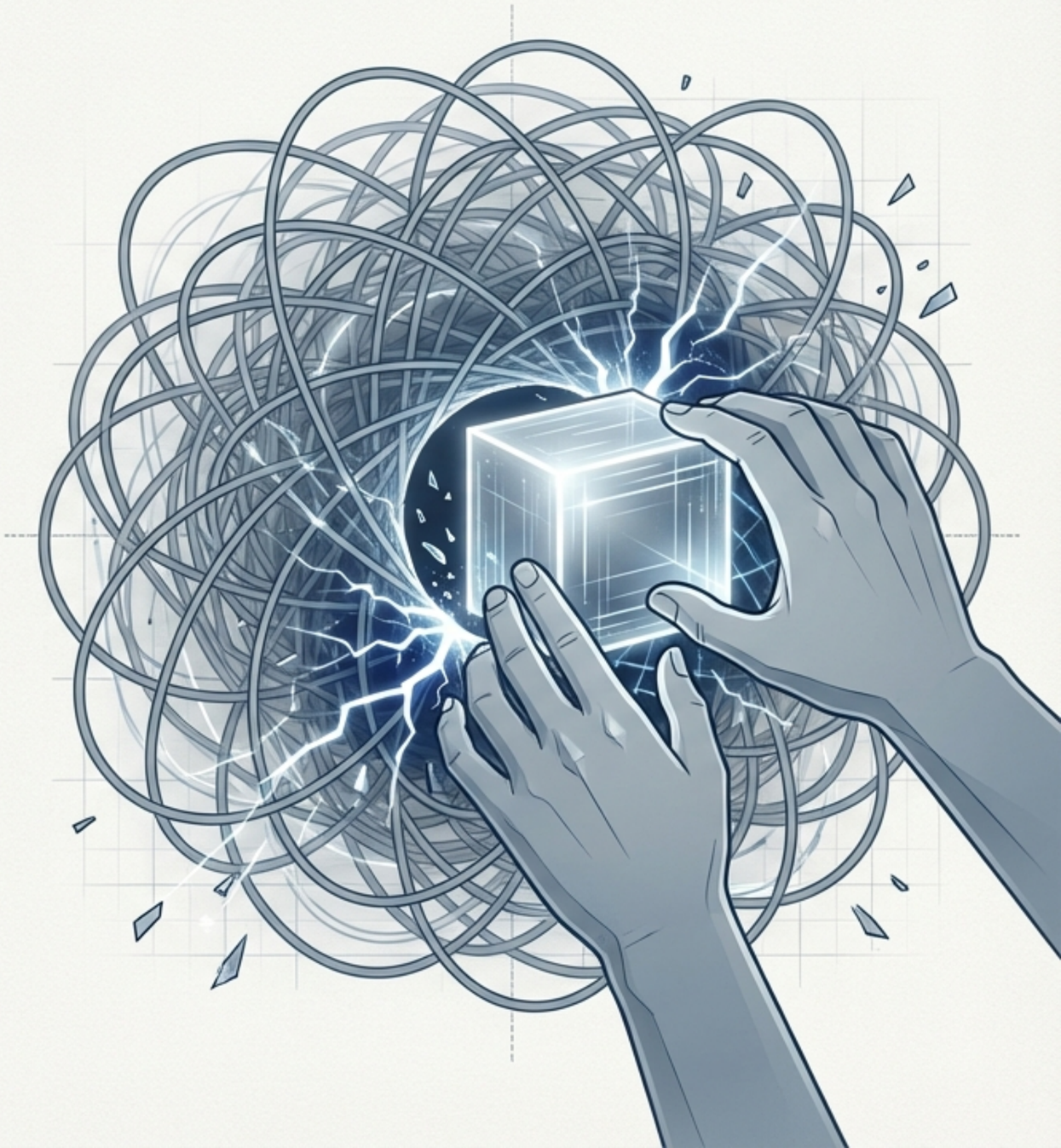
A Domain-Driven Approach for Building Systems That Last



We Often Start Graph Projects by Rushing to the Wrong Tool

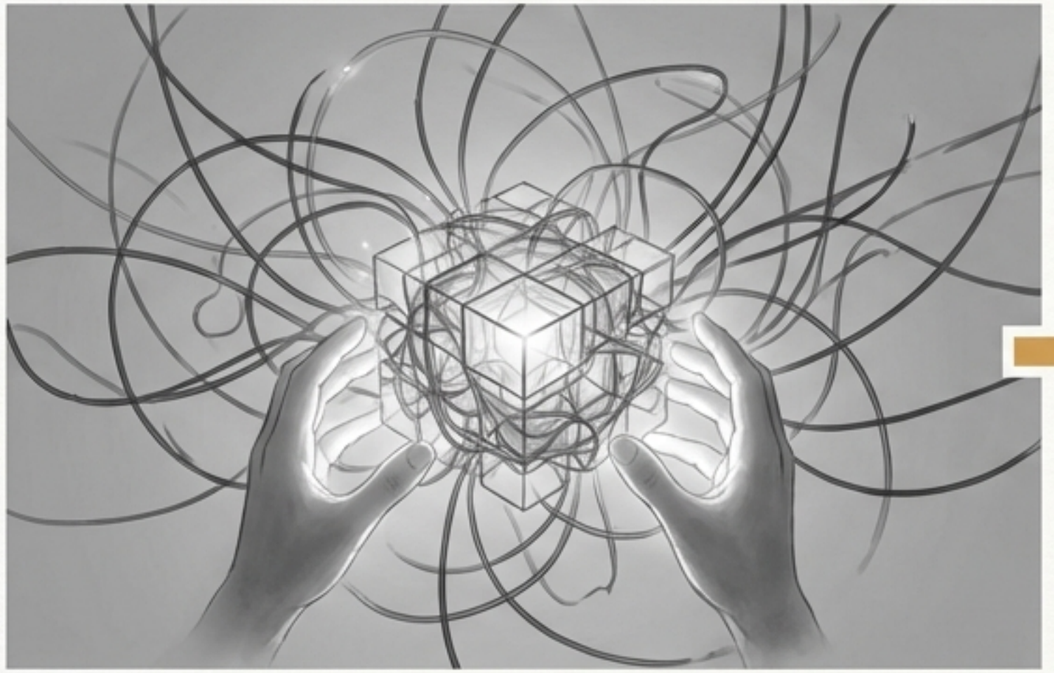
The typical approach is to force all data directly into a generic graph database schema. This ‘one-size-fits-all’ model often leads to a design that doesn’t truly fit the problem, creating unnecessary complexity and shoehorning the domain into an ill-fitting structure. The technology dictates the model, not the other way around.

> *Designing an abstract graph model too early can introduce unnecessary complexity.*

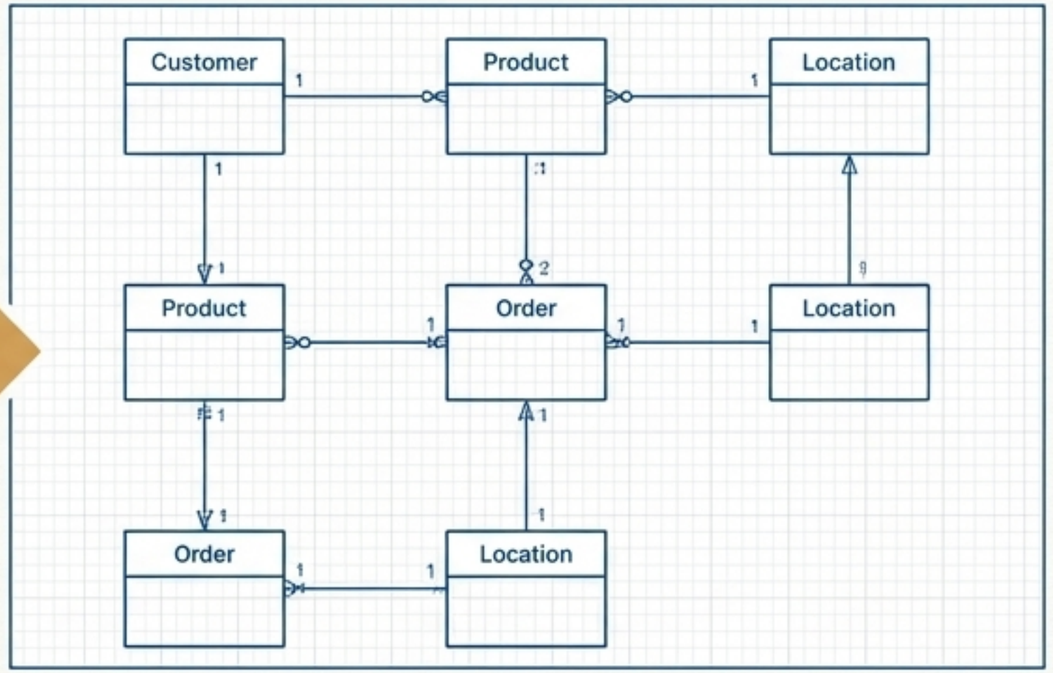


The Solution is a Paradigm Shift: Model the Domain First

Tech-First

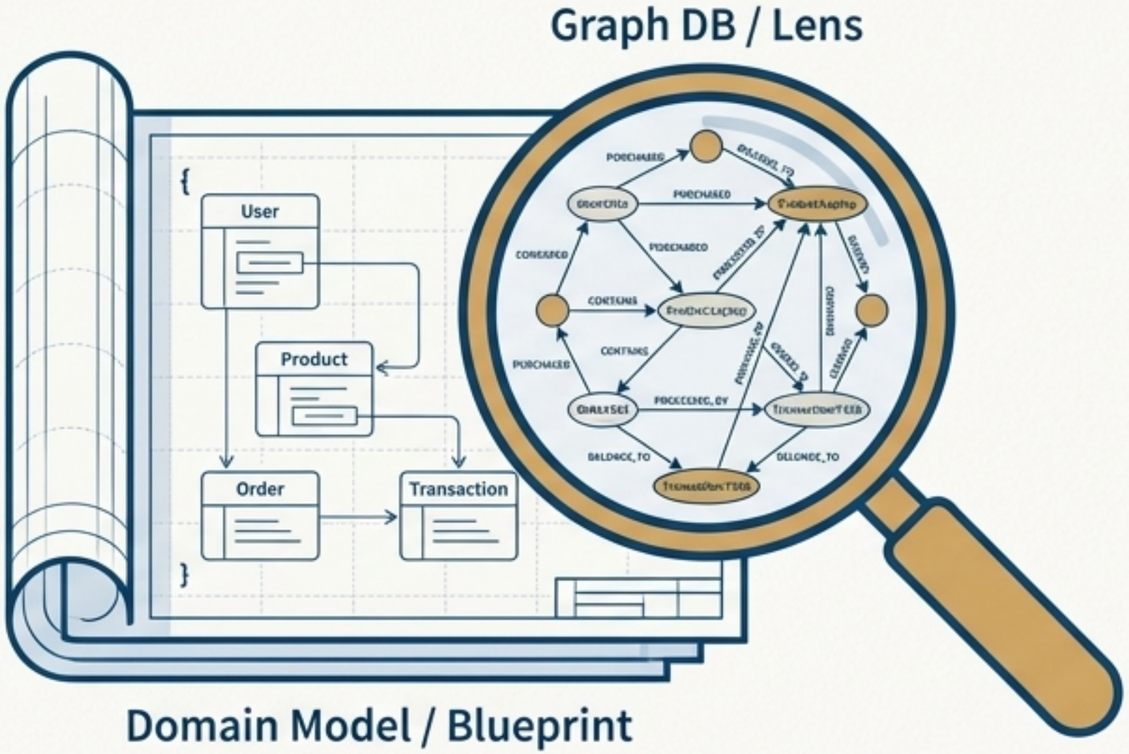


Domain-First



A pragmatic approach focuses on the domain's needs and data first. We design graph structures around real use cases, using native formats like JSON or classes as the primary source of truth. The graph database becomes a powerful tool for analysis, not the foundation of the entire system.

Your Domain Model is the Blueprint; The Graph Database is the Lens



The Blueprint (Source of Truth)

Your data modelled in a format that naturally reflects the domain (JSON, classes, relational tables). It is the carefully designed, authoritative source.

The Lens (Analytical Tool)

The graph database is a powerful tool you use to *view* and *analyse* the blueprint. It helps you see complex relationships and run powerful queries that are difficult in the native format.

Step 1: Model Your Domain Natively, Not in a Generic Graph Schema

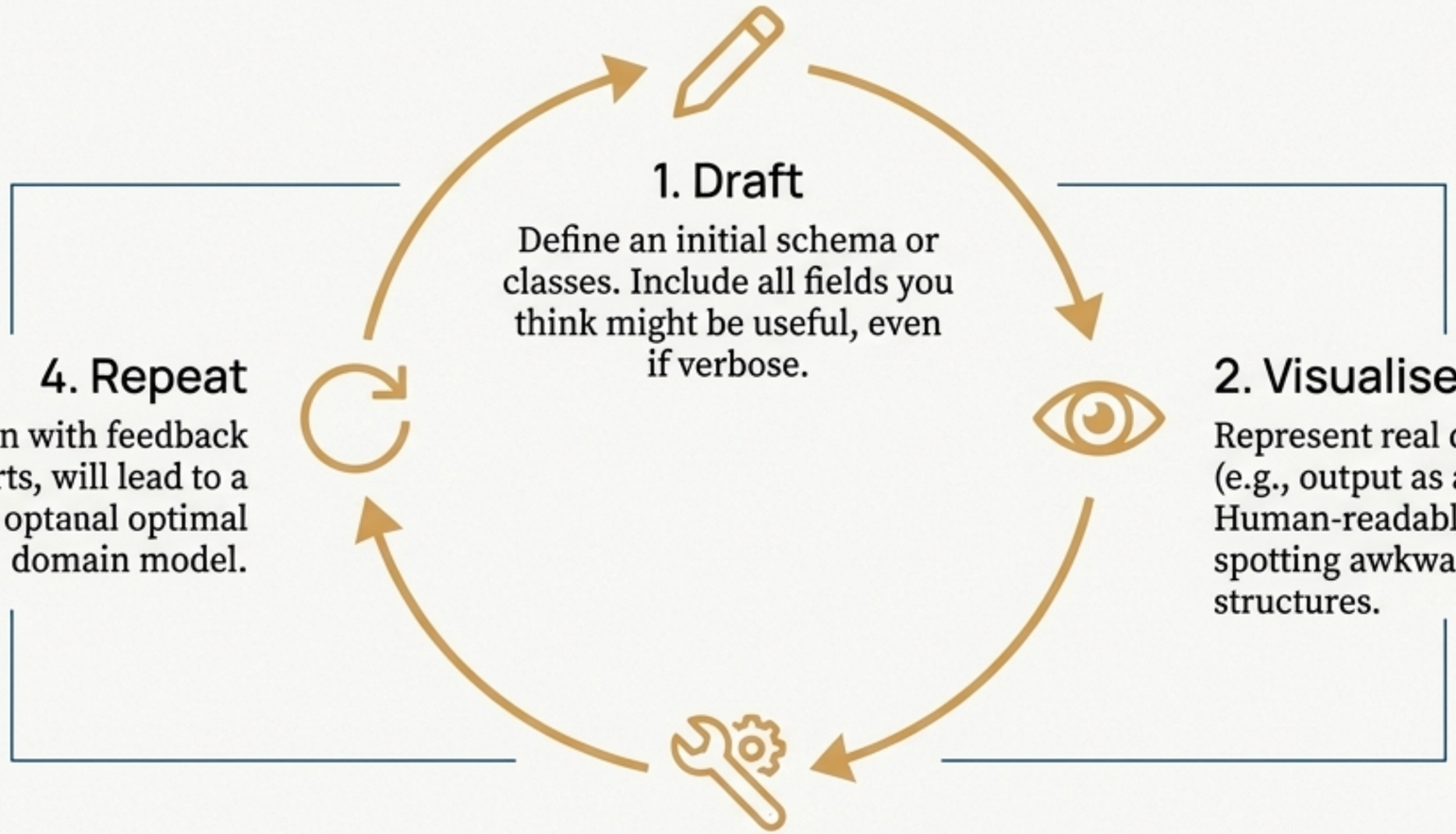
- **Use Native Data Structures First:** Begin with the format that best represents your data (e.g., JSON, Python classes). If modelling source code, use structures that mirror files, classes, and functions.
- **Avoid Premature Abstraction:** Create a specific schema for your problem first. Generalise later if needed. This ensures the model captures exactly what's required, without extraneous concepts.
- **Benefit:** The design remains closely coupled to the problem, making it easier to evolve and refactor.

```python
class Function:
    """Represents a single function in the codebase."""

    def __init__(self, name: str, belongs_to_class: str):
        self.id = f"{belongs_to_class}.{name}"
        self.name = name
        self.belongs_to_class = belongs_to_class
        self.calls: list[str] = [] # List of Function IDs

    def add_call(self, function_id: str):
        self.calls.append(function_id)
```

Step 2: Iterate on the Schema Through Visual Inspection and Refinement



1. Draft

Define an initial schema or classes. Include all fields you think might be useful, even if verbose.

2. Visualise

Represent real data in this format (e.g., output as a JSON structure). Human-readable formats are key for spotting awkward or cumbersome structures.

3. Refine

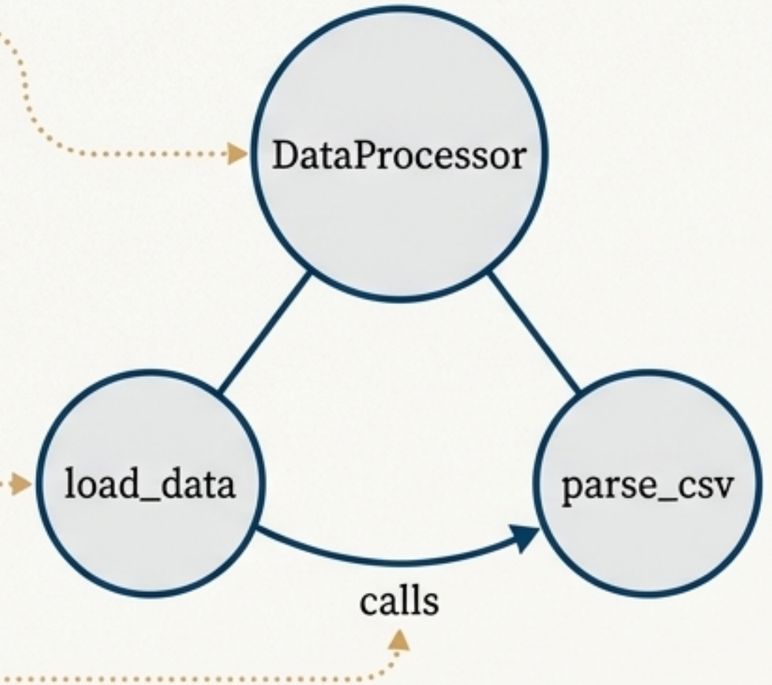
Refactor the schema based on inspection. Remove redundancy, rename for clarity, merge or split structures.

4. Repeat

Multiple passes, often with feedback from domain experts, will lead to a clean, lean, and optanal optimal domain model.

JSON is More Than a Data Format; It's a Human-Readable Graph

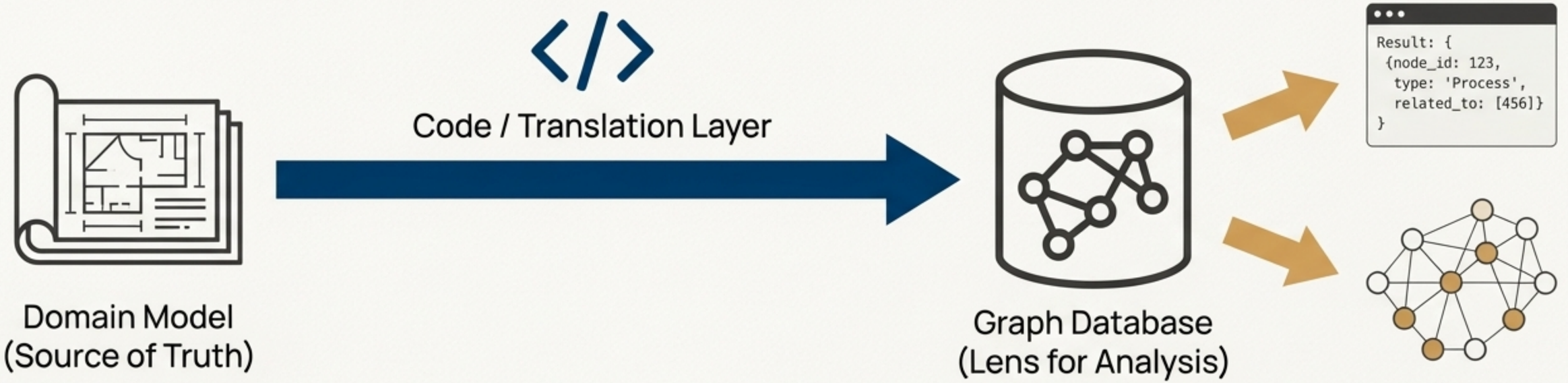
```
{
  "class": "DataProcessor",
  "module": "utils.py",
  "functions": [
    {
      "name": "load_data",
      "id": "utils.DataProcessor.load_data",
      "calls": ["parse_csv", "validate_rows"]
    },
    {
      "name": "parse_csv",
      "id": "utils.DataProcessor.parse_csv",
      "calls": []
  ] }
}
```



JSON and similar nested structures let you see the graph in a tangible, tree-like form, providing invaluable visual feedback. It naturally captures hierarchical data and can represent graph connectivity using IDs or references. You are already working with a graph, just in a more transparent way than a black-box database.

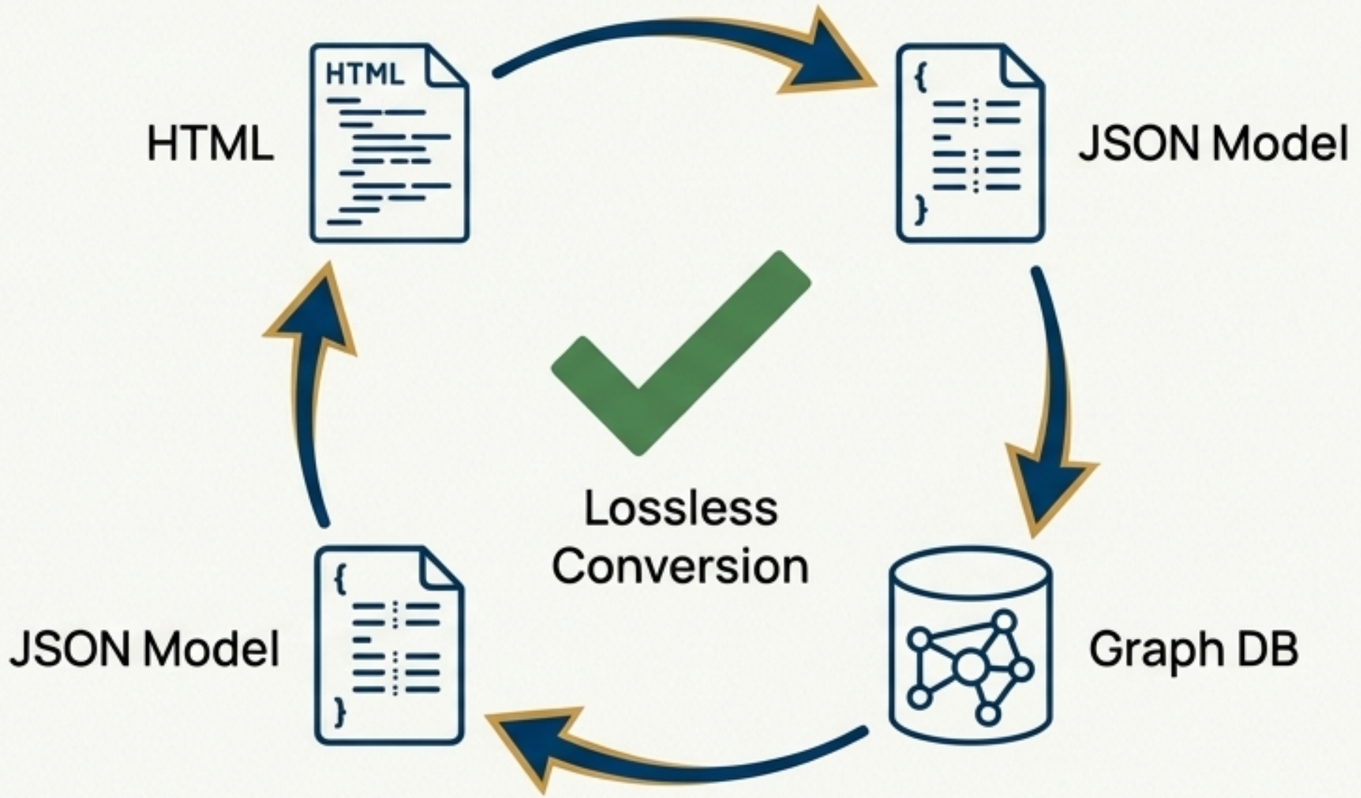
Example from the Source: "In our source code graph project... Seeing it in JSON made it easy to spot if a function was missing a reference to its parent class... This visual clarity is harder to achieve if the data is locked inside a graph database from the start."

Step 3: Project Your Refined Model into a Graph Database for Analysis



- **One-Way Export:** Build a translator that converts your domain objects/JSON into nodes and edges in the graph database (e.g., Neo4j, MGraphDB).
- **Leverage Graph Capabilities:** The database now acts as a powerful lens. Run complex path queries (e.g., multi-hop traversals), find distant relationships, and generate visualisations (Mermaid, D3.js).
- **The Source of Truth Remains:** Your original domain-specific representation remains the authoritative source. You project it into a graph form when needed.

The Gold Standard: Ensuring a Lossless, Round-Trip Data Flow



A robust integration ensures no data is lost in translation.

- **Stable Identifiers are Key:** Every entity in the source model must have a unique ID that is used as the node ID in the graph database. This allows modifications to be mapped back.
- **The Lossless Test:** Can you export to the graph DB and then re-import back to the original format without losing data? This validates that your graph is a true alternate view, not a destructive transformation.
- **Example:** An HTML page was converted to a JSON graph, loaded into a DB, then used to reconstruct the original HTML perfectly, proving all attributes and text were preserved in the JSON model.

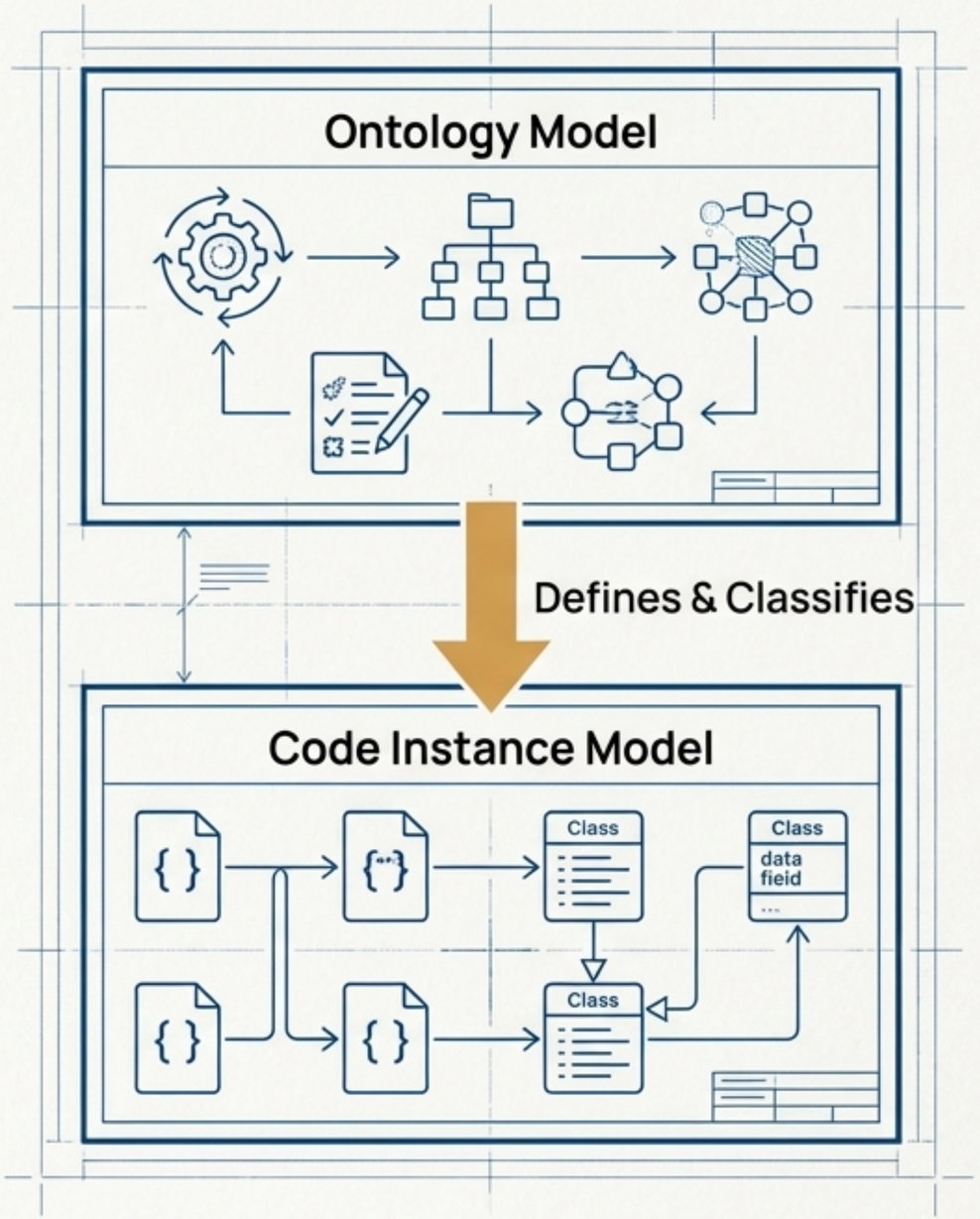
Case Study: Building a Source Code Knowledge Graph

The Challenge: Create a system for analysing a complex codebase to understand dependencies, call flows, and semantic concepts.

The Two Interconnected Graphs:

1. **Semantic Graph (Ontology/Taxonomy):** A meta-graph of concepts describing code (e.g., function categories, relationship types like `CALLS`, `INHERITS`).
2. **Code Instance Graph (Call Flow Graph):** The graph of the actual codebase—modules, classes, functions, and their concrete relationships.

The Naïve Approach (Rejected): Storing everything in one big graph database from the start.



Case Study: Separate, Typed Domain Models Were the Foundation

****The 'Blueprint' in Practice**:**

- **Representation::** Python classes and JSON were used as the primary representation for both the ontology and the code graph.
- **Example (Code Graph):** A `Function` class with fields like `name`, `belongs_to_class`, and `calls = [list of Function IDs]`.
- **Example (Semantic Graph):** An `OntologyNode` class with fields like `concept_name` and `subconcepts`.
- **Key Design Choice':** Each entity was given a stable identifier (e.g., a fully-qualified name) that would serve as the node ID in the graph DB later. This was planned from the beginning.

```
# Python Class Definition
class Function:
    def __init__(self, name, parent):
        self.id = f"{parent}.{name}"
        self.name = name
        self.parent = parent
        self.calls = []
```

```
// Corresponding JSON Output
{
  "id": "DataProcessor.load_data",
  "name": "load_data",
  "parent": "DataProcessor",
  "calls": [
    "DataProcessor.parse_csv",
    "DataProcessor.validate_rows"
  ]
}
```

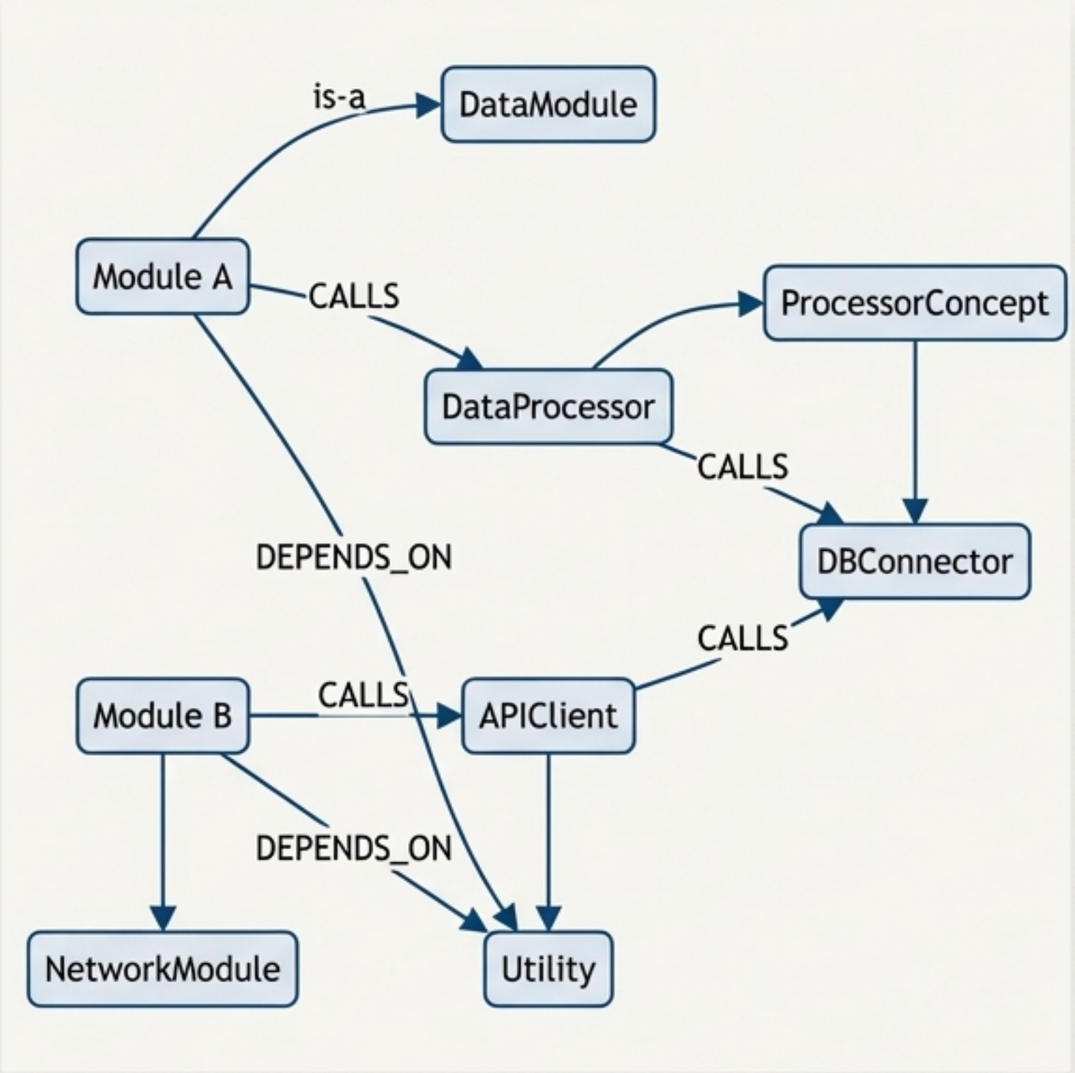
Case Study: Iteration Eliminated Redundancy Before Exporting for Analysis

The Iteration Process in Action:

- Reviewing the JSON output revealed redundancies, like storing a relationship twice (in both the caller and callee). The schema was refactored to store each relationship once.
- Data structures were optimised for access (e.g., using a dictionary for quick lookups by ID instead of a list).

The 'Lens' in Action:

- The refined models were exported to `MGraphDB`. Ontology concepts became nodes with 'is-a' edges. Code elements became nodes (`Function`, `Class`) with edges like `CALLS` and `DEPENDS_ON`.
- **Powerful Queries Enabled:** "Find all functions ultimately related to a certain ontology concept."
- **Visualisations Generated:** Mermaid.js diagrams of module call graphs, D3.js views of the ontology.



Visualisation generated from the projected graph model, enabling rapid analysis of module dependencies.

This Pragmatic Approach Delivers Clarity, Flexibility, and Power

Better Alignment

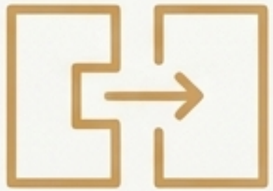
The model is tailored to actual requirements, avoiding the "shoehorning" problem.

Iterative Improvement

Leads to a cleaner, more optimised design before committing to a database schema.

Simplicity and Clarity

Team members can easily understand the model from human-readable JSON or classes.



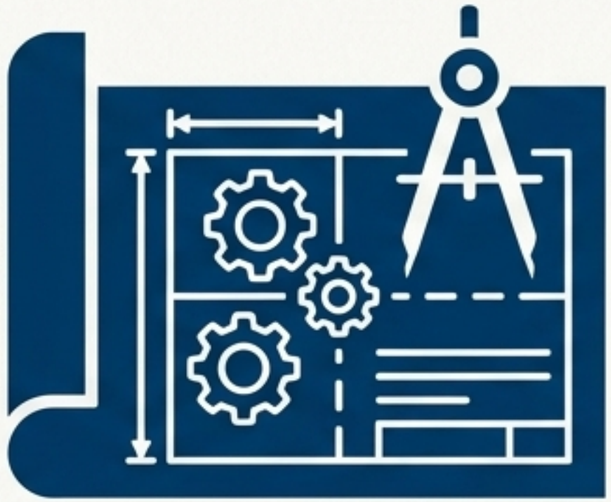
Flexibility to Change Technology

The domain model lives independently. You can swap the graph database without a complete redesign.

Use the Best Tool for the Job

Leverage JSON/classes for editing and validation, and graph databases for complex queries and visualisation.

Your Model is the Asset. The Database is the Tool.



Pragmatic graph development is about being practical and purposeful. Start with the problem and model the data in the simplest, most direct way. Embrace an iterative, domain-driven process. Only then, project the result into a graph database to leverage its unique capabilities. This ensures your graph is a true asset that delivers real value with minimal complexity.